Discrete Logarithm Cryptanalyses: Number Field Sieve and Lattice Tools for Side-Channel Attacks

THÈSE

soutenue le 25 mai 2021

pour l'obtention du

Doctorat de l'Université de Lorraine

(mention informatique)

par

Gabrielle De Micheli

Composition du jury

Président :	Steve KREMER	Directeur de recherche, INRIA, Nancy, France
Rapporteurs :	Martin ALBRECHT	Professor, Royal Holloway, University of London, Royaume-Uni
	Frederik VERCAUTEREN	Associate professor, KU Leuven, Belgique
Examinateurs :	Robert GRANGER	Lecturer, University of Surrey, Royaume-Uni
	Tanja LANGE	Professor, Technische Universiteit Eindhoven, Pays-Bas
	Palash SARKAR	Professor, Indian Statistical Institute, Inde
Encadrants :	Pierrick GAUDRY	Directeur de recherche, CNRS, Nancy, France
	Cécile PIERROT	Chargée de recherche, INRIA, Nancy, France

**Laboratoire Lorrain de Recherche en Informatique et ses Applications — UMR 7503**

i

Remerciements

I would like to express my deep gratitude to my PhD advisors Pierrick Gaudry and Cécile Pierrot for giving me the opportunity to explore in depth the fascinating topic of discrete logarithms and providing invaluable guidance throughout this PhD. I would like to thank them for their patience, enthousiasm and advice which greatly contributed to the concretization of this thesis.

Moreover, I would like to thank Martin Albrecht and Frederik Vercauteran who kindly agreed to review my thesis and provided me with useful comments to improve the quality of this manuscript. I would also like to thank Robert Granger, Steve Kremer, Tanja Lange and Palash Sarkar who agreed to be on my thesis committee.

My thanks also go to all the members of the Penn security group where I spent my first PhD years, and in particular to Nadia Heninger for guiding my first steps in the research world. My PhD journey continued in the Caramba team in Nancy, France and this experience would not have been the same without the incredible kindness of all the members of the Caramba team and the Loria institute. Thank you for sharing your knowledge and passion.

Finally, this work would not have been possible without the encouragement of many friends and family around the world. A particular thanks goes to my cousins in Germany for the many enjoyable weekends spent decompressing away from the computer and to my parents for their constant support.
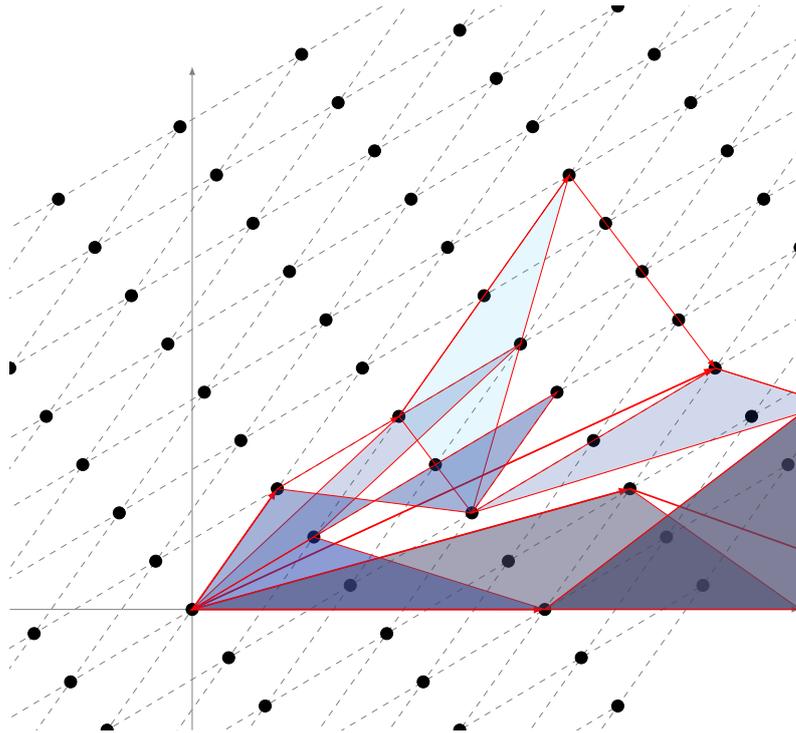
"*From their lofty summits I view my past, dream of the future and, with an unusual acuity, am allowed to experience the present moment.... my vision cleared, my strength renewed.*" Anatoli Boukrev

Contents

x

Introduction

1   Public-key cryptography

Cryptography concerns itself with the problem of exchanging encrypted, meaning unintelligible, messages that only a legitimate receiver can decrypt, hence read. In order to achieve a secure transmission of these messages, a secret key is usually shared between the sender and the receiver. This poses the significant difficulty of securely exchanging the aforementioned secret key.

In the early 1970s, Merkle started deviating from this concept of shared key and his ideas published in 1978 [Mer78] were further carried out in the seminal work of Diffie and Hellman [DH76], *New directions in Cryptography*. In their paper, Diffie and Hellman formalize the notion of public-key cryptography where two mathematically related keys are now generated and used: a public key and a secret key. A message is then encrypted using the receiver's public key. The latter will then be the only one capable of decrypting the message using his corresponding secret key.

Public-key cryptosystems, also known as asymmetric protocols, are all built with the notion of one-way function in mind. The latter corresponds to a function that is easy to compute for any given input but hard to invert. This notion befits the requirements of an asymmetric protocol. Indeed, for a protocol to be secure and efficient, decrypting a message without the secret key should be close to impossible, whereas encrypting a message and decrypting with the secret key should be easy, *i.e.,* only done with simple operations.

It is naturally towards hard mathematical problems that cryptographers have looked to find suitable primitives for their protocols. Historically, two candidates emerged: the multiplication of two prime numbers and modular exponentiation. Inverting these functions consists in factoring an integer and computing a discrete logarithm. The hardness of factoring is at the heart of the well-known and deployed RSA cryptosystem [RSA78]. In this thesis, we will focus on the second candidate: modular exponentiation and its reverse operation, the computation of a discrete logarithm.

Modular exponentiation and discrete logarithm

Modular exponentiation consists in computing the remainder of a Euclidean division of an integer g raised to a power x by a positive integer N, *i.e.,* calculating $g^x \pmod{N}$. This operation is fundamental in computational number theory where it can be seen for example in Fermat's Little Theorem used for primality testing. Modular exponentiation is also extensively used in public-key cryptography where elements of groups such as multiplicative groups of finite fields, $\mathbb{Z}/N\mathbb{Z}$ or the group of rational points of elliptic curves, are often raised to large powers.

For practicality reasons, operations used in cryptographic protocols should be easy and efficient to perform. The attractiveness of modular exponentiation for cryptography comes in part from the simplicity of its computation. Indeed, computing a modular exponentiation can be narrowed down to multiplying g by itself $x - 1$ times and then taking the result modulo N. However, this would be very inefficient in a cryptographic context due to the size of the numbers involved. Hence, in order to construct practical cryptographic schemes that make use of modular exponentiation, the operation must be efficiently performed.

The efficiency of the algorithms that compute modular exponentiation depends on a variety of parameters such as the group considered, the representation of the exponent or the hardware used. Because modular exponentiation is present in many protocols, and is often the most costly operation of the protocol, throughout the years, many optimized algorithms have piled up to improve its computation.

Most algorithms that attempt to optimize modular exponentiation aim at reducing the number of multiplications necessary. One of the oldest and simplest method is the Square-and-Multiply algorithm. The method appeared over 2000 years ago in India [Knu97]. The algorithm operates bit by bit over the bits of the exponent and only performs a multiply operation if that bit of the exponent is 1.

More sophisticated modular exponentiation algorithms make use of different representations of the exponent such as the Non-Adjacent Form (NAF) or its window variant (wNAF). We refer the reader to [Gor98] for a survey on fast exponentiation methods.

While much effort has been put into optimizing the algorithms for modular exponentiation, these optimizations have brought forth exploitable vulnerabilities. Indeed, the operations performed in these algorithms are often dependent on the bitvalues of the exponent. The execution of the code can then generate observable leakage from which information can be deduced about the exponent. The specific character of the information leaked depends on the implementation details of the algorithm and often the hardware itself. Side-channel attacks and in particular cache attacks are the main threats to consider when using a fast modular exponentiation algorithm for a protocol.

The inverse operation of modular exponentiation is the computation of a discrete logarithm. The study of discrete logarithms and related algorithms preceed their use in cryptography. Indeed, as early as the 19th century, Zech's logarithms are used to speed up arithmetic operations in finite fields.

In cryptography, the Diffie-Hellman protocol from the late 70s marked a turning point in the study of discrete logarithms. The more recent use of discrete logarithms in pairing-based protocols which began in the early years 2000, revived the interest in the subject. Concretely, a discrete logarithm is defined as follows.

> **Definition 1** (Discrete logarithm). *Given a finite cyclic group G of order n, a generator $g \in G$, and some element $h \in G$, the discrete logarithm of h in base g is the element $x \in [0, n[$ such that $g^x = h$.*

This definition provides the following problem.

> **Definition 2** (The discrete logarithm problem (DLP)). *Given a finite cyclic group G of order n, a generator $g \in G$, and some element $h \in G$, find x such that $g^x = h$.*

The latter problem is considered hard for most groups G and thus is a promising candidate for public-key cryptography.

> **Question 1.** *Where can discrete logarithms and modular exponentiations be found in cryptography?*

Widely deployed cryptosystems such as the Diffie-Hellman key exchange protocol, ElGamal's encryption protocol or signature schemes such as (EC)DSA base their security on assumptions related to the hardness of the discrete logarithm problem. We describe some of these protocols.

The Diffie-Hellman key exchange protocol [DH76]. The protocol is rather straightforward. Two entities, Alice and Bob, wish to communicate and to do so they must first agree on a secret key. They start by choosing a group G of order n and a generator g of this group, which are now public parameters. Alice then chooses a random element $a \in [0, n[$ and sends the group element g^a to Bob via a public channel. Similarly, Bob chooses $b \in [0, n[$ and sends g^b to Alice. The quantities, g, g^a, g^b are all public. Both Alice and Bob can now compute the quantity

$$s = (g^a)^b = (g^b)^a,$$

which constitutes the shared secret used for future communications.

An attacker who wishes to eavesdrop on any conversation between Alice and Bob must recover the secret value s. Recovering g^{ab} from g, g^a, g^b is known as the computational Diffie-Hellman problem which is closely related to computing discrete logarithms [MW96]. The Diffie-Hellman key exchange protocol became a standard in 2003 (ANSI X9.42) and is used in widely-deployed protocols such as HTTPS/TLS, SSH.

The ElGamal encryption scheme [ElG85]. The ElGamal encryption protocol is closely linked to the Diffie-Hellman key exchange. Alice wants to send an encrypted message to Bob. As for any public-key encryption protocol, Bob needs to generate a pair of keys, a public and a private one which are mathematically related. To do so, he chooses a group G of order n along with a generator g of G. Bob's private key will be an element $b \in [0, n[$ randomly chosen and known only by Bob. The public key are the elements (G, g, g^b).

In order to encrypt a given message m seen as an element of G, Alice will use Bob's public key. Alice starts by choosing a random element $r \in [0, n[$ and computes the quantity $c = m \cdot (g^b)^r$. Alice also computes $c' = g^r$. The ciphertext is thus composed of the two quantities (c, c') which Alice sends to Bob.

Once Bob receives the ciphertext, he can decrypt the message m by using his private key b. Indeed, Bob computes $c(c'^b)^{-1} = m$.

An eavesdropper who intersects the ciphertext (c, c') and wishes to decrypt it would have to solve the computational Diffie-Hellman problem. The ElGamal encryption scheme is extensively used for voting systems [BW14].

The ElGamal signature scheme [ElG85]. Consider $G = (\mathbb{Z}/p\mathbb{Z})^*$ for a prime p. Alice wants to send a message m to Bob and in addition she wants to sign the message in order to authenticate it. Similarly as above, she possesses a pair of secret/public keys (a, g^a) where g is a generator of the group G considered. To generate her signature, Alice selects a random integer $k \in [0, p-1]$ where p is the prime order of G, and such that k and $p-1$ are coprime. She then computes the two quantities $r = g^k \pmod{p}$ and $s = (m - ar)k^{-1} \pmod{p-1}$. Her signature is the pair (r, s) and she sends it to Bob along with the message m.

If Bob wants to verify the validity of the message, he verifies Alice's signature using her public key. From the generation of the signature, we know that $m = ar + sk \pmod{p-1}$. Since Bob knows Alice's public key g^a, we can compare the quantities g^m and $g^{ar}g^{sk}$ modulo p.

Other signature schemes such as the Digital Signature Algorithm (DSA) from 1991 (specifications in FIPS 186-4 [NIS13]), a variant of the ElGamal signature scheme, and its elliptic curve variant ECDSA [JMV01], are also standardized protocols based on the hardness of DLP.

Pairing-based protocols. Pairing-based cryptography is at the heart of numerous security products that are brought to market and research on efficient primitives using them is very active. This is the case in particular in the zero-knowledge area with the applications of zk-SNARKs to smart contracts.

Zero-knowledge proofs allow for a verifier to certify that a prover has knowledge of a secret without revealing information about the secret itself. Zk-SNARKs, Zero-knowledge Succinct Non-interactive Argument of Knowledge, are examples of protocols, widely used in smart contracts for example, that make use of zero-knowledge proofs. Many of these protocols use pairings in their constructions. Evaluating the security of those schemes is thus fundamental. Concretely, a cryptographic pairing is defined as follows.

Definition 3 (Cryptographic pairing). *Consider the finite Abelian groups $\mathbb{G}_1$, $\mathbb{G}_2$, and $\mathbb{G}_T$ of order n. A cryptographic pairing is a map*

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T,$$

which is

- *bilinear, meaning that for any $a, b \in [0, n[$, $P \in \mathbb{G}_1$ and $Q \in \mathbb{G}_2$ we have*

$$e(aP, bQ) = e(P, Q)^{ab},$$

- *non-degenerate, meaning that for any $P \in \mathbb{G}_1$, there exists a $Q \in \mathbb{G}_2$ such that $e(P, Q) \neq 1$ and for any $Q \in \mathbb{G}_2$ there exists a $P \in \mathbb{G}_1$ such that $e(P, Q) \neq 1$,*

- *and computable in polynomial time in the input size.*

The security of pairing-based protocols relies on the hardness of the discrete logarithm problem. In-

deed, in the early 2000s, pairing-based cryptography introduced new schemes such as identity-based encryption schemes [BF01], identity-based signature schemes [CC03], short signature schemes [BLS01] or blind signature schemes used for example in Intel's SGX enclaves (see Chapter 7) whose security is based on pairing-related assumptions that become false if the DLP is broken. To construct a secure protocol based on a pairing, one must thus assume that the DLPs in all three groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are hard.

A natural question arises from the above descriptions of the schemes and the discrete logarithm problem.

> **Question 2.** *What group G should be chosen?*

Evaluating the hardness of DLP has long been an active field of study with many algorithms to solve DLP that appeared over the years. These algorithms vary in their construction and their complexity depends on the group G considered. Cryptography renewed the interest in the discrete logarithm problem over specific groups. In practice, the group G in the definition of the discrete logarithm problem is chosen to be either the multiplicative group of a finite field $\mathbb{F}_{p^n}$ or the group of rational points on an elliptic curve $\mathcal{E}$ defined over a finite field.

Pairing-based cryptography illustrates the need to consider the discrete logarithm problems on both finite fields and on elliptic curves. Indeed, the groups considered for a cryptographic pairing are usually $\mathbb{G}_1$, a subgroup of $\mathcal{E}(\mathbb{F}_p)$, the group of points of an elliptic curve $\mathcal{E}$ defined over the prime field $\mathbb{F}_p$, $\mathbb{G}_2$, another subgroup of $\mathcal{E}(\mathbb{F}_{p^n})$ where we consider an extension field and $\mathbb{G}_T$ a multiplicative subgroup of that same finite field $\mathbb{F}_{p^n}$.

Interestingly enough, the group of rational points on an elliptic curve $\mathcal{E}$ defined over a finite field does not provide any useful representation to speed up the computation of a discrete logarithm. Hence, the best known algorithms to solve DLP over this group are the generic algorithms with square-root complexity.

On the other hand, considering the discrete logarithm problem in finite fields $\mathbb{F}_{p^n}$ have led to considerable improvements in the efficiency of the algorithms that solve it. The nature and complexity of these algorithms depend on the characteristics of the finite field and more precisely the relation between the characteristic p and the extension degree n. The most efficient algorithms to solve DLP in finite fields come from the family of index calculus algorithms. Among those algorithms we have the Function Field Sieve (FFS), the Number Field Sieve (NFS) and its many variants, and the more recent Quasi-Polynomial (QP) time algorithms. An overview of all these algorithms is the focus of Chapter 1.

In 1994, Shor introduced a polynomial time quantum algorithm to compute discrete logarithms. This implies that no scheme relying on the hardness of DLP would be secure in the presence of quantum computers, regardless of the group considered. However, as of today, quantum computers capable of doing large scale computations are non-existent, even though impressive progress has been made in the recent years (see [AAB+19] for a recent 53-qubit machine). Therefore, we will restrict this thesis to the classical setup.

An alternative candidate: lattices

In the past decades, motivated by the upcoming threat of accessible quantum computers, new promising candidates have emerged to construct public-key protocols: lattices, isogenies, error-correcting codes, multivariate polynomials and hash functions. We focus here on lattices.

The study of lattices in mathematics started as early as the 18th century. Mathematicians such as Gauss, Lagrange and Minkowski studied lattices in the context of geometry of numbers and convex geometry, more specifically for the reduction theory of quadratic forms which then led to the well-known Gauss' algorithm. It is not until the 1980s that lattices are studied from a computational point of view and used in areas closer to computer science such as combinatorial optimization and cryptography.

In cryptography, lattices are first used to break cryptosystems. Algorithms such as the Lenstra, Lenstra, Lovász (LLL) algorithm [LLL82] are developed to give approximate solutions to hard lattice problems and are extensively used for cryptanalysis. In 1982, Shamir [Sha82] used the LLL algorithm to break the Merkle-Hellman cryptosystem. In 1996, Coppersmith [Cop96b, Cop96a] proposed a method

that allows to factor an integer n when bits of the factors of n are known, using lattice basis reduction algorithms such as LLL, thus affecting the security of the RSA cryptosystem.

The use of lattices for the design of cryptographic protocols only started in 1996 with the work of Ajtai [Ajt96]. The first cryptosystems to ever use lattices as building blocks are the Ajtai-Dwork [AD97] and NTRU [HPS98] cryptosystems in the late 90s. Today, lattice-based cryptography is a prominent field of study with many lattice-based protocols as candidates in the NIST post-quantum competition. In this thesis we will not consider protocols built on the hardness of lattice-related problems. However, we will use lattice techniques in two different setups:

- we use lattice reduction algorithms to either produce polynomials with small coefficients, see Chapter 3 or to find small roots of modular polynomials, see Part III.

- we use lattice enumeration algorithms, in particular an adaptation of the Schnorr-Euchner algorithm, to speed up the search of algebraic relations in the context of discrete logarithm computations, see Chapter 4.

Background on lattices and related algorithms is thus given in Chapter 2.

Remark 1. *In this thesis, we will talk about lattice sieving and algebraic sieving. These two notions do not refer to the same thing! Algebraic sieving usually refers to a step of the Number Field Sieve (NFS) algorithm whereas lattice sieving such as the Gauss Sieve refers to a lattice algorithm that finds short vectors. Lattice sieving can also be found in the context of NFS. We specify which sieving is being considered when the context is not clear.*

2 Contributions

The goal of this thesis is to answer the following question.

Question 3. *How can we assess the security of protocols in which a modular exponentiation involving a secret is performed?*

The answer to this question is two-fold.

1. Solving the discrete logarithm problem gives direct access to the exponent, hence the secret. Thus, we want to estimate the hardness of DLP in the groups considered by the protocols.

2. Looking at implementation vulnerabilities during fast exponentiation can also lead to the secret exponent. Thus, we also want to assess and study the attacks rendered possible by side-channel leakage.

These two points will shape the structure of this thesis.

2.1 Estimating the hardness of DLP in finite fields

One way of estimating the security of protocols based on the hardness of the discrete logarithm problem is to directly study the complexity of the algorithms that solve the latter. As mentioned above, this is highly dependent on the group considered. In this thesis, we will focus on estimating the hardness of DLP for specific finite fields.

Question 4. *Which finite fields do we focus on and why?*

Finite fields $\mathbb{F}_{p^n}$ are usually separated into three families referred to as small, medium and large characteristic based on the relation between the characteristic p and the extension degree n of the finite field. As of today, the fastest known algorithms to solve DLP are the Quasi-Polynomial time algorithms for finite fields of small characteristic [BGJT14, KW19].

However, in this thesis, we will concern ourselves with finite fields ranging from the boundary between small and medium characteristic up to large characteristic. Indeed, because these families do not provide the fastest known algorithm to solve DLP, they concern most of the finite fields used in practice, for
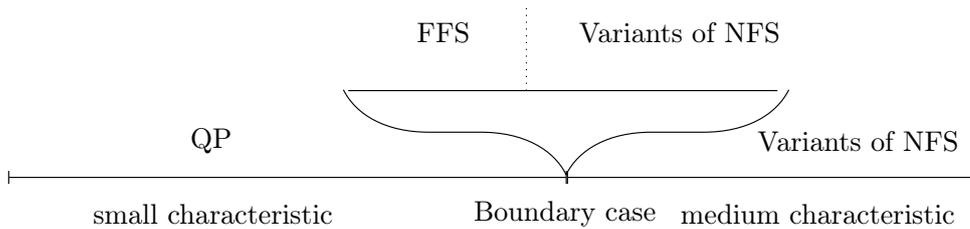
Figure 1: Representation of the finite fields and related algorithms studied in Chapter 3.

example in pairing-based protocols. Hence, estimating the hardness of DLP for these families have a significant impact on our understanding of the security of widely-deployed protocols.

Our first motivation concerns the security of pairing-based protocols. If we want a pairing to be secure, we want to balance the complexity of the square-root algorithm that computes discrete logarithms in the relevant subgroup of the elliptic curve considered, and the complexity of the best-known algorithm that solve DLP in the finite field. This brought us to study the algorithms from the index-calculus family mentioned above at the boundary between small characteristic finite fields and medium characteristic ones. The asymptotic complexities of these algorithms at this boundary case was, up to this thesis, non-existent in the literature. This study also allowed us to provide precise crossover points between these numerous complexities. This will be the focus of Chapter 3, illustrated in Figure 1. From this analysis, we were finally able to provide further information about security parameters for pairing-based protocols. More precisely, this chapter answers the following question.

Question 5. *Asymptotically what finite field $\mathbb{F}_{p^n}$ should be considered in order to achieve the highest level of security when constructing a pairing?*

In this chapter, we give optimal values for characteristic p and extension degree n, also taking into account the so-called ρ-value of pairing constructions. Surprising fact, we were also able to distinguish some special characteristics that are asymptotically as secure as characteristics of the same size but without any special form. The following paper summarizes our results.

1. **Asymptotic complexities of discrete logarithm algorithms in pairing-relevant finite fields**, with Pierrick Gaudry and Cécile Pierrot, published in the proceedings of Crypto 2020.

Another way of having better security estimates is to run large scale experiments with variants of the Number Field Sieve. Indeed, the Number Field Sieve algorithm gave rise to many variants, each attempting to lower the asymptotic complexity of the original algorithm. One of these variants is the Tower Number Field Sieve (TNFS). The latter exploits the algebraic structure of towers of number fields. Despite the fact that in theory the variant is more than promising, no implementations and thus record computations had been done using TNFS, up to this thesis.

A major obstacle to an efficient implementation of TNFS is the collection of algebraic relations where equations between small elements of number fields must be found. The case of TNFS is more complex than NFS as this relation collection happens in dimension greater than 2. This requires the construction of new sieving algorithms which remain efficient as the dimension grows. In Chapter 4, we overcome this difficulty by considering a lattice enumeration algorithm which we adapt to this specific context. We also consider a new sieving area, a high-dimensional sphere, whereas previous sieving algorithms (for dimension 2 and 3) considered an orthotope.

This allowed us to perform the first record computation of a discrete logarithm with TNFS in a 521-bit finite field $\mathbb{F}_{p^6}$. The target finite field $\mathbb{F}_{p^6}$ chosen is of the same form than finite fields used in recent zero-knowledge proofs in some blockchains. This record computation was announced in February 2021.

2. **Discrete logarithm in GF(p^6) with Tower NFS** with Pierrick Gaudry and Cécile Pierrot, announced in the NumberTheory mailing list. Related article in submission process.

Details about both the implementation and the computation are provided in Chapter 5. As can be seen in Table 1, our algorithm is much faster than existing high-dimensional sieving algorithms despite the larger dimension and the larger finite field.

Parameters	[GGMT17]	[MR21]	This thesis
Algorithm	NFS	NFS	TNFS
Field size (bits)	422	423	521
Sieving dimension	3	3	6
Sieving time	201,600	69,120	**23,300**

Table 1: Comparison of the relation collection step in core hours with [GGMT17] and [MR21] for $\mathbb{F}_{p^6}$.

Both these works contribute to estimating the hardness of DLP in finite fields by studying the asymptotic complexities of the relevant algorithms and providing an actual record computation with TNFS. The considerations made on the security of pairings should complete the practical estimates found in the literature and hopefully point the cryptanalysts to the right parameter choices. The practical performance of TNFS with our new sieving algorithm is promising and indicates that larger finite fields could be reached in a reasonable amount of time. In general, record computations provide further indications on the gap between the recommended key sizes for protocols based on DLP and what is computationally feasible.

2.2 Exploiting implementation vulnerabilities from fast exponentiation

The security of deployed protocols not only relies on the hardness of the underlying mathematical problem but also on the implementation of the algorithms involved.

Vulnerable implementations of fast modular exponentiation have often been the target of microarchitectural side-channel attacks where secret information is recovered by creating observable contentions between different CPU execution units. In particular, timing attacks exploit variations in execution time which is common in modular exponentiation algorithm.

In Chapter 6, we present an overview of the techniques known to recover secret keys from partial information. The leaked information, usually a certain amount of bits of a secret element of the protocol, is illustrated in Figure 2.

Knowledge of partial information (bits)

Figure 2: Example of representation of leaked partial information from a side-channel attack.

Numerous techniques to recover secret keys from partial information exist depending on the nature of the information recovered by the side-channel attack and on the specificities of the algorithm used. This chapter presents the most useful techniques along with a comprehensive classification on what is known to be efficient for the most commonly encountered scenarios in practice. We focus on the widely-used algorithms which are the most popular targets for attacks, *i.e.,* RSA, (EC)DSA and (elliptic curve) Diffie-Hellman. Our results appear in the following paper.

3. **Recovering cryptographic keys from partial information, by example**, with Nadia Heninger. Available on Eprint:Report 2020/1506.

The techniques presented in Chapter 6 have often led to real-world attacks on deployed protocols. In this thesis, we focus on two of these techniques which rely on lattice constructions: the Hidden Number Problem and the Extended Hidden Number Problem.

In Chapter 7, we investigate the security of the Intel implementation of the Extended Privacy ID (EPID) protocol, a remote authentication and attestation protocol. We identify an implementation

weakness that leaks information via a cache side channel. This leaked information allows us to mount a lattice-based approach for solving the Hidden Number Problem, which we adapt to the zero-knowledge proof in the EPID scheme, extending prior attacks on signature schemes. This work shows that a malicious attestation provider can use the leaked information to break the unlinkability guarantees of EPID. We also give experimental evidence that the lattice attack can still succeed even when a small number of erroneous traces are included. These results are reported in the following paper.

4. **CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks**, with Fergus Dall, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi and Yuval Yarom, published in IACR Transactions on Cryptographic Hardware and Embedded Systems,Volume 2, 2018.

We finally focus on the security of the ECDSA protocol when the nonce k used in the signing algorithm as modular exponent is expressed in wNAF form. In Chapter 8, we reinvestigate the construction of the lattice used in the Extended Hidden Number Problem (EHNP). We find the secret key with only 3 signatures, thus reaching a known theoretical bound, whereas best previous methods required at least 4 signatures in practice. Given a specific leakage model, our attack is more efficient than previous attacks, and for most cases, has better probability of success. We also provide a first error-resilience analysis of EHNP. This work is described in the following paper.

5. **A Tale of Three Signatures: Practical Attack of ECDSA with wNAF**, with Cécile Pierrot and Rémi Piau, published in the proceedings of Africacrypt 2020.

By considering real-world targets such as EPID in Intel's architecture and the widely-deployed ECDSA algorithm, we show throughout these works that even if the right parameters are considered for the discrete logarithm problem to remain hard enough to solve for cryptographic purposes, attacks can come from vulnerable implementations of modular exponentiation. In order to truly evaluate the security of deployed public-key cryptosystems, one must concurrently consider the threats from the mathematical primitive itself and the implementation of the algorithms.

The contributions and the organization of the thesis are summarized in Figure 3.
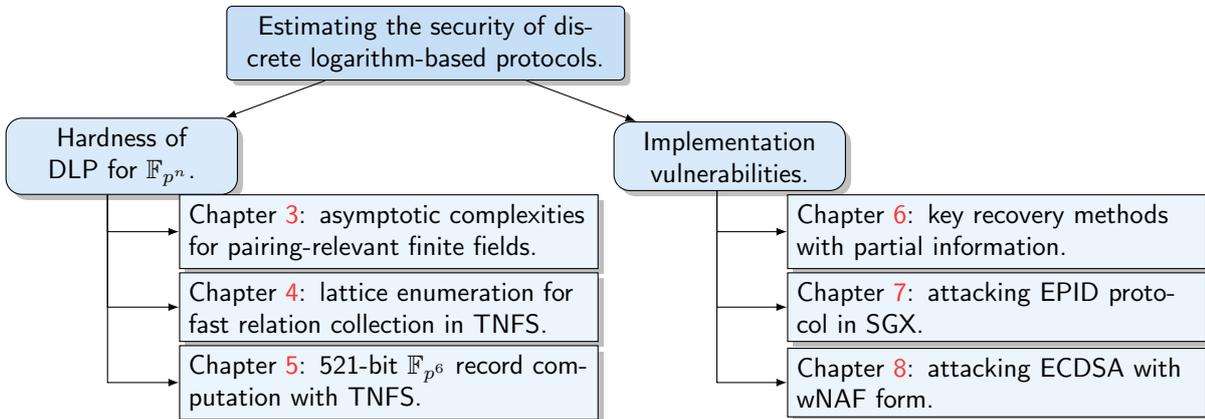
Figure 3: Organization of the contributions of this thesis.

2.3   Other contributions

Overstretched NTRU is a variant of NTRU with a large modulus. Recent lattice subfield and subring attacks have broken suggested parameters for several schemes. There are a number of conflicting claims in the literature over which attack has the best performance. These claims are typically based on experiments more than analysis.

In this work, we argue that comparisons should focus on the lattice dimension used in the attack. We give evidence, both analytically and experimentally, that the subring attack finds shorter vectors and

thus is expected to succeed with a smaller dimension lattice than the subfield attack for the same problem parameters, and also to succeed with a smaller modulus when the lattice dimension is fixed.

Because the thematic of this work is outside the main theme of this thesis, we do not include it in the manuscript. The following paper summarizes these results.

6. **Characterizing overstretched NTRU attacks**, with Nadia Heninger and Barak Shani. At Mathcrypt 2018, published in Journal of Mathematical Cryptology, Volume 14, no. 1, 2020.

Part I

Preliminaries

Chapter 1

Overview of algorithms for the discrete logarithm problem

The discrete logarithm problem (DLP) plays a central role in public-key cryptography as the security of many protocols relies on its hardness. In order to evaluate the difficulty of solving DLP in particular groups, one must analyze the complexity of the known algorithms that solve DLP in these groups.

In this chapter, we present both heuristic and provable algorithms that solve the discrete logarithm problem. The construction of these algorithms as well as their complexities highly depend on the group considered.

In Section 1.1, we present some of the oldest examples of algorithms that appeared in a context that preceded the importance of discrete logarithms in cryptography. These algorithms are known as generic algorithms and have a square root complexity in the size of the group. The most well-known are the Baby-step Giant-step algorithm and Pollard's algorithms. The complexity of these algorithm can be further reduced when the order of the group is composite using Pohlig-Hellman's reduction. These algorithms work for arbitrary groups as they make no use of possible specificities of the group structure.

However, the particular structure of groups such as multiplicative groups of finite fields can lead to more efficient algorithms. This is the case of the family of index calculus algorithms, presented in Section 1.2. The latter also find their origin much earlier than their use in cryptography but have since become the starting point of many efficient algorithms with sub-exponential complexity.

Lastly, Section 1.3 provides a general description of the index calculus algorithms we will concern ourselves with in this thesis: the Function Field Sieve (FFS), the Number Field Sieve (NFS) and its numerous variants. These algorithms are among the fastest known algorithms to solve the discrete logarithm problem in finite fields and led to quite a few record computations in the past decades.

This chapter restricts itself to the classical setup thus we will not adventure ourselves in the quantum world despite Shor's polynomial time quantum algorithm. Moreover, we do not consider the Area-Time (AT) complexity model [BL14].

Contents

1.1 Generic algorithms for the discrete logarithm problem

In this section, we present algorithms that solve the discrete logarithm problem in *generic groups*.

Following the work of Babai and Szemeredi [BS84] and Nechaev [Nec94], Shoup formally introduced in 1997 [Sho97] the *generic group model*. The latter provides a rigorous definition of a generic algorithm, *i.e.*, an algorithm that does not exploit any particular structure of the group considered. More precisely, it relies on the assumption that group operations are done by means of an oracle and elements of the group are encoded as bitstrings.

Note that this model is far from a realistic representation of a group that could be used in cryptography. Indeed, the encoding of group elements is generally distinguishable from random bitstrings (for example, the identity element of a multiplicative group is usually denoted 1 or (0, 1, 0) for elliptic curves etc). Thus one should simply view the generic group model as a mean to ensure that the group considered in the discrete logarithm problem does not have any properties that could be used to solve the problem in a more efficient way. We refer to [KM07] for a discussion of Shoup's model and an explanation of the differences between the generic group model and the more well-known (in cryptography) random oracle model.

For the remaining of the section, let G be a finite cyclic multiplicative Abelian group with generator g and order n. We denote $t \in G$ the target element whose discrete logarithm x defined as $x = \log_g t$ we are looking for.

1.1.1 Pohlig-Hellman's reduction

In 1978, Pohlig and Hellman [PH78] introduced a method that reduces the problem of solving DLP in the entire group to solving it in smaller instances. The main idea of their algorithm is to reduce the computation within the group G of order n to subgroups of prime order. Of course, this implies that n is composite and that its factorization is known.

Let n decompose into $n = \prod_{i=1}^{k} p_i^{f_i}$ for distinct primes p_i and integers $f_i > 0$. Then there exists a unique subgroup $G_i \subset G$ of order $n_i = n/p_i^{f_i}$ for all $i = 1, 2, \cdots, k$. Let $g_i = g^{n_i}$ and thus $g_i^{p_i^{f_i}} = (g^{n_i})^{p_i^{f_i}} = g^n = 1$. The order of g_i is then exactly $p_i^{f_i}$ by Lagrange's theorem and g_i is a generator of G_i.

For each factor $p_i^{f_i}$, we consider the target $t_i = t^{n_i} \in G_i$ and one can solve the discrete logarithm problem in the corresponding subgroup G_i by finding

$$x_i = \log_{g_i} t_i \pmod{p_i^{f_i}}.$$

Because $x \equiv x_i \pmod{p_i^{f_i}}$ for all $i = 1, 2, \cdots, k$ from the definitions given above, after having found all the x_i, the Chinese remainder theorem (CRT) can be used to recover the solution x. This reduction to subgroups is illustrated in Figure 1.1.

$$t = g^x \pmod{n} \text{ with } n = p_1^{f_1} \cdots p_k^{f_k}$$

$$x_1 \equiv x \pmod{p_1^{f_1}} \quad \cdots \quad x_i \equiv x \pmod{p_i^{f_i}} \quad \cdots \quad x_k \equiv x \pmod{p_k^{f_k}}$$

$$x \pmod{n} \text{ with CRT}$$
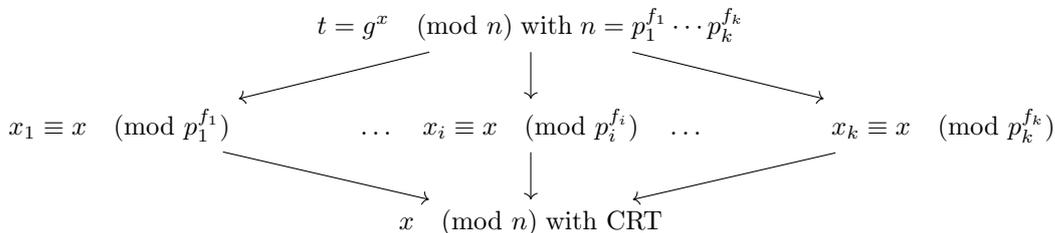
Figure 1.1: Pohlig-Hellman decomposition in subgroups.

It then remains to focus on the discrete logarithm in each of the subgroups.

Question 6. *How does one compute $x_i = \log_{g_i} t_i$ in each of the subgroups G_i?*

The idea is again to reduce the problem further down to prime order subgroups. Let us consider one of the subgroups G_i of order $p_i^{f_i}$. The goal is to find $x_i = \log_{g_i} t_i$, where $t_i = g_i^{x_i} \in G_i$. We remove the subscript i for ease of reading. We know the element x is smaller than the order of the subgroup, that is p^f, and hence can be re-written as $x \equiv x_0 + x_1 p + \cdots + x_{f-1} p^{f-1} \pmod{p^f}$ with $x_i \in [0, p)$ for all $i = 1, 2, \cdots, f - 1$. One can then note that the equation $t = g^x$ raised to the power p^{f-1} becomes

$$t^{p^{f-1}} = g^{x p^{f-1}} = g^{x_0 p^{f-1}},$$

the second equality coming from the expression of $x \pmod{p^f}$. This intrinsically means that $x_0 = \log_{g^{p^{f-1}}} t^{p^{f-1}}$, *i.e.*, the coefficient x_0 is the discrete logarithm of $t^{p^{f-1}}$ in the subgroup generated by $g^{p^{f-1}}$, which is of prime order p.

More generally, all the x_i can be seen as discrete logarithms in prime subgroups. Indeed, the coefficients $x_1, x_2, \cdots, x_{f-1}$ can be found in a similar way by induction. Assume we have found all the coefficients up to x_k and we are looking to determine x_{k+1}. In a similar fashion as before, we have

$$\left(t g^{-\sum_{i=0}^{k} x_i p^i} \right)^{p^{f-k-2}} = g^{x_{k+1} p^{k+1}}$$

from which we obtain x_{k+1} as the discrete logarithm of the left-hand-side of the equation in the same subgroup generated by $g^{p^{f-1}}$ of order p.

Pohlig-Hellman's algorithm thus reduces the problem of computing a discrete logarithm in a group of order n to computing discrete logarithms in prime order subgroups. The complexity of Pohlig-Hellman's reduction is $O\left(\sum_i f_i (\log n + \sqrt{p_i}) \right)$. It narrows down to the sum of the costs of computing discrete logarithms in all the subgroups. Computing the discrete logarithm in the prime order subgroups can be done with algorithms using a square-root complexity as we will now explain. Note that the Pohlig-Hellman reduction assumes that the order n of the group is composite. This is not always the case. If the order of the group G is not composite, the complexity becomes $O(\sqrt{n})$. On the other hand, if n is composite, the complexity is bounded by the square root of the size of the largest p_i in the prime factorization of n.

We now present two generic algorithms with no assumption on the compositeness of n and hence ignoring Pohlig-Hellman's reduction.

1.1.2 Baby-step Giant-step algorithm

The Baby-step Giant-step algorithm due to Shanks [Sha71] is a simple deterministic algorithm based on a time-memory tradeoff as we will now explain.

Let $m = \lceil \sqrt{n} \rceil$. Then there exists integers $0 \leq a, b < m$ such that the exponent x can be written as $x = a + bm$. The target element can thus be re-written as $t = g^x = g^{a+bm}$, which is equivalent to $g^{bm} = tg^{-a}$. The algorithm starts by pre-computing a (large) list of "baby steps": $L = \{(a, tg^{-a}) : 0 \leq a < m\}$. It is important here to note that this list not only must be stored in memory, but also in an adequate way as to efficiently be able to determine whether an element is in it or not. To do so, one usually uses hash tables.

The algorithm proceeds by computing larger steps known as "giant steps" calculating g^{bm} for b values in $[0, m)$. Each time a giant step is computed, the algorithm compares with values from the baby steps and checks in the list L if there exists a pair (a, tg^{-a}) such that $tg^{-a} = g^{bm}$. If this is the case, the algorithm found a value $x = a + bm$ such that $t = g^x$. This algorithm is illustrated in Figure 1.2.

The complexity of the algorithm depends on the cost of computing the baby steps and the giant steps. The baby step list L contains m elements and thus requires m group operations. The algorithm computes at most m giant steps and thus the algorithm performs at most $2m$ group operations. The time complexity is then $O(\lceil \sqrt{n} \rceil)$. Note that this algorithm also has a space complexity of $O(\lceil \sqrt{n} \rceil)$ due to the necessity of storing the elements of L. Improvements on the number of groups operations can be found in [GWZ17, BL13].
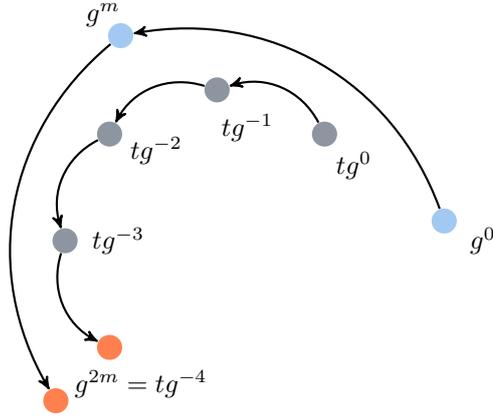
Figure 1.2: Example of baby steps (grey) and giant steps (blue). A collision is found for $g^{2m} = tg^{-4}$, thus $x = 2m + 4$.

1.1.3 Pollard algorithms

A major issue with the Baby-step Giant-step algorithm is the space requirement. In 1978, Pollard [Pol78] suggested a randomized algorithm known as the Pollard rho algorithm which requires the same number of group operations as Baby-step Giant-step but with the advantage of a $O(1)$ space requirement. The idea is quite similar to Baby-step Giant-step and relies on the possibility of a collision happening between exponents of elements of the group.

Let $a_i, b_i \in [0, n)$ be integer values for all indices $i = 0, 1, \cdots, n-1$ that characterize elements of G of the form $t^{a_i} g^{b_i}$. The algorithm uses a random walk to detect a collision, meaning two indices, say j and k, such that $t^{a_j} g^{b_j} = t^{a_k} g^{b_k}$. More precisely, let $f : G \to G$ be a function used to define the random walk: $t^{a_{i+1}} g^{b_{i+1}} = f(t^{a_i} g^{b_i})$. We refer to [Tes00] for details on constructions of the function f. If we assume that the values $t^{a_i} g^{b_i}$ behave as uniformly distributed random elements in G, the birthday paradox will ensure a collision happens after approximately $\sqrt{n}$ steps. The random walk is illustrated in Figure 1.3.
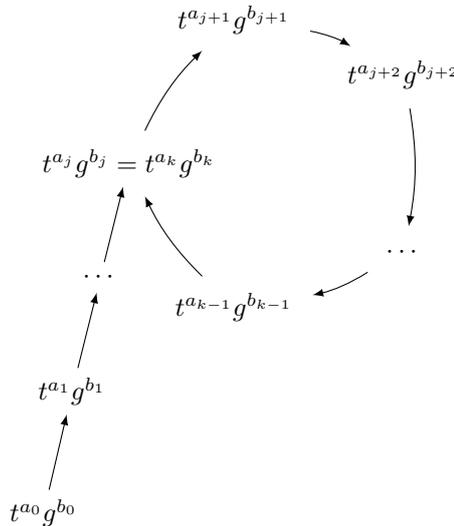
Figure 1.3: Random walk in Pollard rho algorithm.

Once a collision has occured, the discrete logarithm can be recovered as $x = (b_j - b_k)/(a_k - a_j)$ (mod n), where one must have $a_j \neq a_k$ otherwise the algorithm fails. Pollard rho's algorithm is probabilistic whereas the Baby-step Giant-step algorithm is deterministic.

The complexity is given by the number of steps in the random walk, thus is $O(\sqrt{n})$. We also mentioned the $O(1)$ space requirement as the major practical advantage over the previous algorithm. The latter comes from the use of Floyds' cycle-finding algorithm (also known as the Tortoise and the Hare algorithm) which compares values $t^{a_i} g^{a_i}$ with $t^{a_{2i}} g^{a_{2i}}$ and thus only requires to store two values at any given time.

Solving DLP in a small interval. Pollard also proposed a variant known as the Pollard kangaroo algorithm [Pol78] used when the discrete logarithm is known to lie in a small interval. The complexity remains the same as Pollard Rho's but the variant is computationally faster. This variant can be used for example in the case where some bits of the discrete logarithm are known thus reducing the search space. We will discuss and illustrate the use of Pollard's kangaroo algorithm in the context of key recovery with partial information in Chapter 6.

Both Pollard's algorithms allow to use less memory which is a significant advantage for practical computations. Another major improvement concerns the scalability of these algorithms. Van Oorschot and Wiener in [OW99] showed how to parallelize Pollard's algorithms using the method of distinguished points.

We presented two generic algorithms that solve DLP in a group G with complexity $O(\sqrt{|G|})$. We have also seen that this complexity can be further reduced if the order of the group is composite and becomes $O(\sqrt{p})$ where p is the size of the largest prime factor of n using Pohlig-Hellman's reduction. It is natural to wonder whether a better algorithm exists for the generic group model.

Question 7. *Can one find an algorithm for generic groups that has complexity lower than $O(\sqrt{p})$?*

The answer is no. Shoup used his generic group model to prove that the discrete logarithm problem in a generic group of order p runs in $\Omega(\sqrt{p})$ group operations.

1.2 Index Calculus methods

The square root complexity of the generic algorithms presented above can be reduced if the group considered has a particular structure that can be exploited. In particular, it was noted in [MW68, Mil75] that the specific structure of multiplicative groups of finite fields could help in the design of more efficient algorithms to solve DLP. This is the case of index calculus algorithms.

The index calculus method is an alternative to generic group algorithms first investigated in the work of Kraitchik [Kra22] in the 1920s. This family of algorithms has led to a long succession of efficient proven or heuristic algorithms often used for cryptanalysis, the most well-known being the Number Field Sieve.

1.2.1 Index calculus algorithms

In the context of finite field DLP computations, an index calculus algorithm takes as input the multiplicative group of a finite field $\mathbb{F}_{p^n}$ and a target element $t \in \mathbb{F}_{p^n}^*$. It then outputs the discrete logarithm of t in base g, the generator of the group, in sub-exponential time.

Any index calculus algorithm follows three main steps: a relation collection step, a linear algebra step and finally an individual logarithm step.

Collecting relations. This first step consists in collecting multiplicative relations between elements of $\mathbb{F}_{p^n}^*$. These multiplicative relations are of the form

$$\prod_{f \in \mathbb{F}_{p^n}^*} f^{e_f} = g^m$$

where e_f and m are integer values. The goal is then to obtain the logarithms of these elements. To do so, the multiplicative relations given above are transformed into additive linear relations

$$\sum_{f \in \mathbb{F}_{p^n}^*} e_f \log_g(f) \equiv m \pmod{p^n - 1}.$$

These relations form a system of equations where the unknowns are the $\log_g f$. The relation collection step stops when enough relations are collected. The algorithm requires as many relations as unknowns in order to uniquely solve the system.

Question 8. *Concretely how do we collect relations?*

Relation collection has significantly evolved since the first index calculus algorithms in the 1970s. The first algorithm proposed by Adleman [Adl79] in prime fields of order p computed $g^a \pmod{p}$ until an integer a was found such that g^a factored into a product of small primes. This simple method led to a first sub-exponential complexity for the discrete logarithm in prime fields.

However, in more recent algorithms, relations are computed more efficiently by using a commutative diagram. The idea is then to represent the equality characterizing a relation by taking two different paths in the diagram which lead to the same result. This is illustrated in Figure 1.4. At the top of the diagram, a ring R is adequately chosen depending on the context.

$$
\begin{array}{ccc}
 & R & \\
 \swarrow^{\psi_1} & & \searrow^{\psi_2} \\
 I_1 & & I_2 \\
 \searrow_{\varphi_1} & & \swarrow_{\varphi_2} \\
 & \mathbb{F}_{p^n}^* &
\end{array}
$$
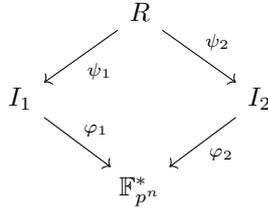
Figure 1.4: Commutative diagram to collect relations.

The maps $\psi_1, \psi_2, \varphi_1, \varphi_2$ are morphisms and because the diagram is commutative, for $x \in R$ we have

$$\varphi_1(\psi_1(x)) = \varphi_2(\psi_2(x)) \in \mathbb{F}_{p^n}^*.$$

A relation is found if a given $x \in R$ produces two equal products in $\mathbb{F}_{p^n}^*$ and the quantities $\psi_1(x)$ and $\psi_2(x)$ decompose into small enough factors in the intermediate rings I_1, I_2. Let $\mathcal{F}_1$ and $\mathcal{F}_2$ be two small subsets of "small" elements of I_1, I_2. Then, we have a relation if

$$\psi_1(x) = \prod_{f \in \mathcal{F}_1} f^{e_f} \text{ and } \psi_2(x) = \prod_{f' \in \mathcal{F}_2} f'^{e_{f'}}.$$

Finally, the relation is expressed as

$$\varphi_1(\psi_1(x)) = \prod_{f \in \mathcal{F}_1} \varphi_1(f^{e_f}) = \prod_{f' \in \mathcal{F}_2} \varphi_2(f'^{e_{f'}}) = \varphi_2(\psi_2(x)) \in \mathbb{F}_{p^n}^*,$$

from which we have

$$\sum_{f \in \mathcal{F}_1} e_f \log(\varphi_1(f)) \equiv \sum_{f' \in \mathcal{F}_2} e_{f'} \log(\varphi_2(f')) \pmod{p^n - 1}.$$

Definition 4 (Factor basis). *We will define the factor basis as $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$.*

Remark 2. *The exact definition of the elements of a factor basis depends on the context and in particular on the definitions of I_1 and I_2. We will further discuss this when detailing the algorithms. Moreover, the term factor basis often refers directly to the images by φ_1, φ_2 of the elements of $\mathcal{F}$.*

Linear algebra.   Once enough relations are collected, they form a linear system of equations where the unknowns are the discrete logarithms of the images by φ_1, φ_2 of factor basis elements. Solving the system gives an element of the kernel of the matrix of relations and the discrete logarithms of the elements of the factor basis are found up to a constant, meaning there exists $c > 0$ such that $c \log_g \varphi_i(f)$ is a solution for $f \in \mathcal{F}_i$, $i = 1, 2$. This constant c can easily be found if g is in the factor basis. Indeed, if $g \in \mathcal{F}_i$, one solution will be $c \log_g \varphi_i(g) = x$ and since $\log_g \varphi_i(g) = 1$, we explicitly recover the constant c. By solving this system, one then gets all the logarithms of the elements of the factor basis.

> **Question 9.** *How do we efficiently solve this system?*

It is important to note that the system is sparse, meaning that many coefficients in the matrix are equal to zero. Indeed, when considering a row of the matrix, *i.e.*, a relation $\sum_{f \in \mathcal{F}_1} e_f \log(\varphi_1(f)) = \sum_{f' \in \mathcal{F}_2} e_{f'} \log(\varphi_2(f'))$ (mod $p^n - 1$), most of the coefficients $e_f, e_{f'}$ are equal to zero as many elements of the factor basis will not necessarily appear in the factorization of $\psi_1(x)$ and $\psi_2(x)$.

The sparsity of the matrix allows to use efficient algorithms to solve the system with quadratic complexity instead of cubic, such as block Wiedemann algorithm [Cop94].

Individual logarithm phase.   We now have the discrete logarithms of all the elements of the factor basis. However, the desired output is the discrete logarithm of an arbitrary (larger) element t of the group $\mathbb{F}_{p^n}^*$. The general idea is then to decompose t into a product of elements of the factor basis. Indeed, if $t = \prod_{f \in \mathcal{F}} f^{\alpha_f}$ and we are looking for x such that $t = g^x$, then we have

$$x \equiv \sum_{f \in \mathcal{F}} \alpha_f \log_g(f) \pmod{p^n - 1}.$$

> **Question 10.** *What is the complexity of index calculus algorithms?*

The main idea of index calculus algorithms is to decompose elements of the group into a product of bounded elements. The complexity of the algorithm thus relies on the probability of elements of the group to easily decompose into smaller parts. This brings forth the following definition.

> **Definition 5** (B-smoothness). *An integer (resp. a polynomial) is said to be B-smooth if it decomposes into prime factors smaller than B (resp. into irreducible factors of degree smaller than B.)*

Let $\psi(x, y)$ denote the number of positive integers smaller than x which are y-smooth. What interests us more than evaluating the number $\psi(x, y)$ is the probability of a random number smaller than x to be y-smooth. This probability is given by the quantity $\psi(x, y)/x$.

Mathematical results on the probability of smoothness of random integers were given first by Dickman [Dic30] and then by Canfield, Erdős and Pomerance in [CEP83]. Dickman provided a first asymptotic formula for $\psi(x, y)$ by showing that for any fixed $u > 0$

$$\lim_{x \to \infty} \psi(x, x^{1/u})/x = \rho(u).$$

The function $\rho(u)$ is commonly known as Dickman's rho function and is defined as the unique solution to the differential equation

$$u \rho'(u) + \rho(u - 1) = 0,$$

for $u > 1$ and $\rho(u) = 1$ for $0 \leq u \leq 1$.

The following theorem states an even stronger result.

Theorem 1 (Canfield, Erdős, Pomerance). *If $\epsilon > 0$ and $3 \leq u \leq (1-\epsilon) \log x / \log \log x$, then $\psi(x, x^{1/u}) = x u^{-u + o(u)}$.*

This smoothness result for random integers can be adapted to polynomials over finite fields [PGF98].

Theorem 2 (Panario, Gourdon, Flajolet). *The number of y-smooth monic polynomials of degree x over $\mathbb{F}_{p^n}$ is given by*

$$\Psi_{p^n}(x, y) = (p^n)^x \rho\left(\frac{x}{y}\right)\left(1 + O\left(\frac{\log x}{y}\right)\right),$$

where ρ is the Dickman's rho function.

The probability of smoothness for both integers and polynomials can be summarized in the following corollary.

Corollary 1. *The probability that a random integer (resp. random polynomial) smaller than x (resp. of degree smaller than x) is y-smooth is given by $u^{-u+o(1)}$ where $u = \log x/\log y$ (resp. $u = x/y$).*

Part of the complexity analyses of index calculus algorithms rely on these probabilities. This introduces however a major heuristic: we assume in these analyses that the elements created in the relations behave as random integers (or polynomials). The only fully proven algorithms were given by Pomerance in 1987 [Pom87] and more recently by Kleinjung and Wesolowski [KW19].

Let us now go back to the complexity analysis of index calculus methods. The algorithm being divided into three steps, the total cost of the algorithm is the sum of the costs of these three steps which we now explain.

Relation collection.   The cost of this step mainly corresponds to the cost of finding enough relations, *i.e.,* finding $|\mathcal{F}|$ relations. Let P_B be the probability of finding a B-smooth element. Finding enough relations has a cost of $|\mathcal{F}|/P_B$ operations. One should also take into account the cost of testing for B-smoothness, but we will see in more detailed analyses that it is negligible with respect to other costs.

Linear algebra.   The cost of linear algebra is the cost of solving the linear system of equations formed by the relations. As we mentioned before, there exist quadratic time algorithms that solve sparse systems and thus the cost of this step is $|\mathcal{F}|^2$ operations.

Individual logarithm step.   Finally, the cost of the individual logarithm step narrows down to the cost of splitting the target element t into B-smooth factors and thus corresponds to $1/P_B$ operations. A more rigorous explanation leading to a more accurate cost will be given in Chapters 4 and 5 but the cost remains negligible compared to the two other costs.

In order to facilitate the expression of the complexity of these steps and the total complexity of the algorithm, the smoothness results and the quantities involved in the complexity analyses are usually restated using a notation known as the L_{p^n}-notation that arises from the smoothness probabilities. This notation allows in particular to visualize these algorithms as sub-exponential time algorithms, meaning not as fast as polynomial time algorithms but better than exponential time ones. The terminology is explained in the following section.

1.2.2   Small, medium and large characteristics

The index calculus method gave rise to many algorithms that solve DLP over finite fields $\mathbb{F}_{p^n}$ and their complexities vary depending on the relation between the characteristic p and the extension degree n. Three families of finite fields emerged in which different variants of the index calculus method perform best.

The L_{p^n}-notation is used to both define these families of finite fields and express the complexities of the algorithms. The latter is defined as

$$L_{p^n}(\alpha, c) = \exp((c + o(1))(\log p^n)^\alpha (\log\log p^n)^{1-\alpha}),$$

for $0 \leqslant \alpha \leqslant 1$, some constant $c > 0$ and $o(1) \to 0$ when $p^n \to \infty$. The constant c is ignored if the context does not require it specifically.

Let us first consider the L_{p^n}-notation as a means to characterize finite fields. The characteristic p of the finite field is expressed using the L_{p^n}-notation. Now, with $p = L_{p^n}(\alpha, c)$, we talk of large characteristic

finite fields if $\alpha > 2/3$, medium characteristic finite fields if $\alpha \in (1/3, 2/3)$ and small characteristic finite fields if $\alpha < 1/3$. These families are illustrated in Figure 1.5.

One can see that the definition matches the intuition. A binary field $\mathbb{F}_{2^n}$ is a finite field said of small characteristic. Indeed, we can write $p = 2 = L_{2^n}(0, c) = \exp(c+o(1))(\log\log 2^n)$ with $c = \log 2/\log\log 2^n$. On the other hand, a prime field $\mathbb{F}_p$ is said to be of large characteristic as one can write $p = L_p(1, c) = \exp(c+o(1))(\log p)$ with $c = 1$. Note however that the definition is considered asymptotically and is thus harder to conceptualize for fixed parameters.

Figure 1.5: Three families of finite fields with examples. For the examples, we use the approximation $\alpha \approx \frac{\log\log p}{\log\log p^n}$. The notation p_a means a prime p of bitsize a.

These families are separated by two boundary cases: $p = L_{p^n}(1/3, c)$ and $p = L_{p^n}(2/3, c)$.

Question 11. *What does it mean to be at a boundary case?*

Let us consider the first boundary case as it will be our study-case in Chapter 3. This boundary case corresponds to the equation $p = L_{p^n}(1/3, c)$ which implies $\log p = \log(p^n)^{1/3}\log\log(p^n)^{2/3}$, ignoring the constant. As we are mainly trying to convey an intuition, let us ignore the $\log\log$ term and have $(\log p)^3 \approx \log(p^n)$. Thus, the first boundary case represents the family of finite fields such that $n \approx (\log p)^2$.

This L_{p^n}-notation is not only used to characterize families of finite fields, but it is also used to express the complexities of the algorithms that solve DLP in finite fields. The notion of sub-exponential time complexity can be seen with the definition. Indeed, when $\alpha \to 0$, we have $L_{p^n}(\alpha, c) \approx \exp(c\log\log p^n) \approx (\log p^n)^c$ which implies a polynomial time complexity in the size of the input, the finite field. On the other hand, when $\alpha \to 1$, we get $L_{p^n}(\alpha, c) \approx p^{cn}$ which is exponential time. When $0 < \alpha < 1$, the $L_{p^n}(\alpha, c)$ complexity is thus sub-exponential.

Question 12. *Which algorithms are more efficient in each family of finite fields?*

The most efficient algorithms that solve DLP in finite fields all come from the index calculus method presented above. Among the most well-known algorithms we have the Number Field Sieve and its variants, the Function Field Sieve and the more recent family of Quasi-Polynomial time algorithms (QP). We briefly go over the history of these algorithms to answer Question 12 and refer to Section 1.3 for more details.

Small characteristic. In the case of small characteristic, Coppersmith [Cop84] gave a first $L_{p^n}(1/3)$ algorithm in 1984. Introduced in 1994 by Adleman, the Function Field Sieve [Adl94] also tackles the DLP in finite fields of small characteristic. The algorithm relies on the arithmetic of function fields. In 2006, Joux and Lercier [JL06] proposed a description of FFS which does not require the theory of function fields, and Joux further introduced in [Jou13] a method, known as pinpointing, which lowers the complexity of the algorithm.

In 2013, after a first breakthrough complexity of $L_{p^n}(1/4 + o(1))$ by Joux [Jou14], a heuristic Quasi-Polynomial time algorithm was designed [BGJT14] by Barbulescu, Gaudry, Joux and Thomé. Variants were explored in the following years [GKZ14, GKZ18, JP14a, JP19] with two different goals: making the algorithm more practical, and making it more amenable to a proven complexity. We mention two key ingredients. First, the so-called zig-zag descent allows to reduce the problem to proving that it is possible to rewrite the discrete logarithm of any degree-2 element in terms of the discrete logarithms of linear factors, at least if a nice representation of the finite field can be found. The second key idea is to replace

a classical polynomial representation of the target finite field by a representation coming from torsion points of elliptic curves. This led to a proven complexity in 2019 by Kleinjung and Wesolowski [KW19]. To sum it up, the Quasi-Polynomial algorithms outperform all previous algorithms both theoretically and in practice in the small characteristic case.

Medium characteristic. For finite fields of medium characteristics, NFS and its variants remain as of today the most competitive algorithms to solve DLP. Originally introduced for factoring, the NFS algorithm was first adapted by Gordon in 1993 to the discrete logarithm context for prime fields [Gor93]. A few years later, Schirokauer [Sch00] extended it to finite fields with extension degrees $n > 1$. In [JLSV06], Joux, Lercier, Smart and Vercauteren finally showed that the NFS algorithm can be used for all finite fields. Since then, many variants of NFS have appeared, gradually improving on the complexity of NFS. The extension to the Multiple Number Field Sieve (MNFS) was originally invented for factoring [Cop93] and was then adapted to the discrete logarithm setup [Mat03, BP14]. The Tower Number Field Sieve (TNFS) [BGK15] was also introduced in 2015.

When n is composite, this variant has been extended to exTNFS in [KB16, KJ17]. The use of primes of a special form gives rise to another variant called the Special Number Field Sieve (SNFS) [JP14b]. Most of these variants can be combined with each other, giving rise to MexTNFS and S(ex)TNFS. The complexities of all these variants are summarized in Table 1.1.

Large characteristic. The variants used in medium characteristic can also be used in large characteristic, the best complexity being achieved using special primes. The complexities are reported in Table 1.1.

Specificity	Algorithm	medium characteristic	2nd boundary	large characteristic
None	NFS	96	48	64
	MNFS	89.45	45.00	61.93
	TNFS	–	–	64
	MTNFS	–	–	61.93
Composite n	exTNFS	48	48	64
	MexTNFS	45.00	45.00	61.93
Special p	SNFS	$64\left(\frac{\lambda+1}{\lambda}\right)$	$\star$	32
	STNFS	–	–	32
Composite n and special p	SexTNFS	32	$\star$	32

Table 1.1: Best complexities of NFS and its variants in medium and large characteristics. The complexities are all given as $L_{p^n}(1/3, (c/9)^{1/3})$ and we report the value c. We write $-$ when a variant for specific sizes of finite fields leads to a complexity greater than $L_{p^n}(1/3)$. For the 2nd boundary case, we report the best achieved constant as the complexity is non-monotonic in this case. For $\star$, these values also depend on another parameter λ and thus we refer to [JP14b, KB16] for details.

Figure 1.6 summarizes the performance of these algorithms in relation to the three families of finite fields defined above.

> **Question 13.** *What happens at the two boundary cases?*

One can see from Figure 1.6 that the boudary case $p = L_{p^n}(2/3)$ separates two families where essentially the same algorithms perform best: the Number Field Sieve and its variants. The precise analysis of the complexity of these algorithms at this boundary case has been studied as a particular case in most papers that study these complexities in either the medium or the large case.

However, the boundary case $p = L_{p^n}(1/3)$ looks more suspicious as algorithms using different techniques overlap in this area. Quasi-Polynomial time algorithms and FFS come from the small characteristic family whereas NFS and its variants start performing well in medium characteristic. A precise analysis of the complexities of the aforementioned algorithms at this boundary case is the focus of Chapter 3.

We now describe the general setup of FFS and NFS as well as its variants. We do not detail the family of Quasi-Polynomial time algorithms as they appear very little in this thesis.
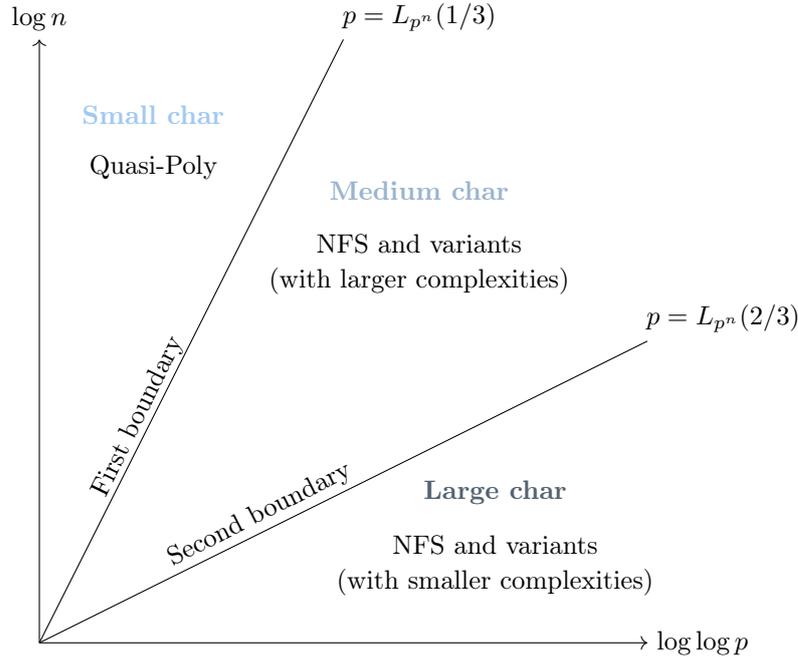
Figure 1.6: Families of finite fields with best performing algorithms.

1.3 The general setting of FFS, NFS and its variants

In this section, we introduce a general description of the Number Field Sieve algorithm, its variants and the similar Function Field Sieve algorithm. We focus on the overall structure of the algorithms following the steps introduced for index calculus methods.

We will provide substantially more technical details for the description of one variant, the Tower Number Field Sieve, in Chapter 4, as we provide a first implementation of it as well as a first record computation.

Before introducing this general framework, let us look at the definitions of the mathematical objects that give their names to these algorithms.

Definition 6 (Number field). *A number field is a finite degree extension of $\mathbb{Q}$. It is defined as $K = \mathbb{Q}[x]/(f)$ where f is an irreducible polynomial in $\mathbb{Q}[x]$.*

The degree of the number field K is the degree n of the polynomial f. The elements of K are called algebraic numbers.

When replacing "number" by "function", we have the following similar definition.

Definition 7 (Function field). *Let p be a prime number. A function field is a finite extension of the rational function field $\mathbb{F}_p(\iota)$. Let f be an irreducible polynomial in $\mathbb{F}_p(\iota)[x]$. Then, the function field defined by f is the field $\mathbb{F}_p(\iota)[x]/(f)$.*

Elements of a function field are called algebraic functions.

The main objects we will concern ourselves with in number and function fields are ideals and in particular their norms. We recall the definition of the norm of an ideal.

Definition 8 (Norm of ideal). *Let K be an algebraic number or function field and $\mathcal{O}_K$ its ring of integers. The norm of a non-zero ideal I is $N_K(I) = |\mathcal{O}_K/I|$.*

Computing the norms of ideals will play a central role in complexity analyses of index calculus algorithms. In the case of number fields, we will exploit the relation between the norm and the resultant. If K is a number field associated to an irreducible polynomial $f \in \mathbb{Z}[x]$, meaning $K = \mathbb{Q}[x]/(f)$ and θ is a root of f, then for $\phi \in \mathbb{Z}[x]$, the norm of $\phi(\theta) \in K$ satisfies

$$N(\phi(\theta)) = \pm f_n^{\deg \phi} \operatorname{Res}(\phi, f),$$

where f_n is the leading coefficient of f.

In the case of function fields, the norms are computed with a bivariate resultant and correspond to polynomials.

Interestingly enough, the same arithmetic theory applies to finite extensions of $\mathbb{F}_p(\iota)$ and finite extensions of $\mathbb{Q}$. This allows NFS and FFS to share many similarities and thus to be presented in a combined setup.

1.3.1   Overview of the algorithms: a general presentation

Consider a ring R that is either $\mathbb{Z}$ in the most basic NFS, a number ring $\mathbb{Z}[\iota]/(h(\iota))$ in the case of Tower NFS, or $\mathbb{F}_p[\iota]$ in the case of FFS. The algorithm starts by selecting V distinct irreducible polynomials $f_i(x)$ in $R[x]$ in such a way that there exist maps from $R[x]/(f_i(x))$ to the target finite field $\mathbb{F}_{p^n}$ that make the diagram commutative. This leads to the construction given in Figure 1.7 which should remind the reader of Figure 1.4. In order to have these maps, the polynomials are selected such as to share a common irreducible factor of degree n.

Question 14. *Why do we need a common irreducible factor of degree n?*

Let us consider the simple case where $R = \mathbb{Z}$. Let $I(x)$ be an irreducible factor of degree n in $\mathbb{F}_p[x]$ shared by all the polynomials f_i. The target finite field can be expressed as $\mathbb{F}_{p^n} \cong \mathbb{F}_p[x]/(I(x))$ and let m be a root of $I \pmod{p}$. Then $I(m) = 0$ in $\mathbb{F}_{p^n}$ and thus $f_i(m) = 0 \pmod{p}$. Hence all the polynomials f_i share a common root m in $\mathbb{F}_{p^n}$ and the maps from $\mathbb{Q}[x]/(f_i(x))$ to the target finite field $\mathbb{F}_{p^n}$ simply correspond to an evaluation in m. Using the same notations as in Figure 1.4, we have the maps

$$\psi_i : \quad \begin{array}{ccc} \mathbb{Z}[x] & \to & \mathbb{Q}[x]/(f_i(x)) \\ x & \mapsto & \theta_i \end{array}$$

where θ_i is a root of f_i and

$$\varphi_i : \quad \begin{array}{ccc} \mathbb{Q}[x]/(f_i(x)) & \to & \mathbb{F}_{p^n} \\ \theta_i & \mapsto & m \end{array}$$

A polynomial $\phi(x) \in \mathbb{Z}[x]$ can be written as $\phi(x) = f_i(x)Q_i(x) + P_i(x)$ with $Q_i, P_i \in \mathbb{Z}[x]$. It is first mapped to $\phi(\theta_i) = f_i(\theta_i)Q_i(\theta_i) + P_i(\theta_i) = P_i(\theta_i)$ in $\mathbb{Q}[x]/(f_i(x))$ and then to $f_i(m)Q_i(m) + P_i(m) = P_i(m)$ in $\mathbb{F}_{p^n}$ since $f_i(m) = 0 \pmod{p}$. A relation is thus an equation in $\mathbb{F}_{p^n}$ of the form $P_i(m) = P_j(m) \pmod{p}$ for $i \neq j$.

Based on this construction, the discrete logarithm computation follows the same steps as any index calculus algorithm:

- Relation collection: we collect relations built from polynomials $\phi \in R[x]$ of degree $t - 1$, and with bounded coefficients. If R is a ring of integers, we bound their norms, and if it is a ring of polynomials, we bound their degrees. The intermediate fields considered are then either number fields or function fields.

  For number fields: Because there exists no unique factorization of elements of a number field, the elements of the factor basis are chosen to be prime ideals in the ring of integers of the number fields.

$$R[X]$$

$$R[x]/(f_1(x)) \quad R[x]/(f_2(x)) \quad \ldots \quad R[x]/(f_i(x)) \quad \ldots \quad R[x]/(f_{V-1}(x)) \quad R[x]/(f_V(x))$$

$$\mathbb{F}_{p^n}$$
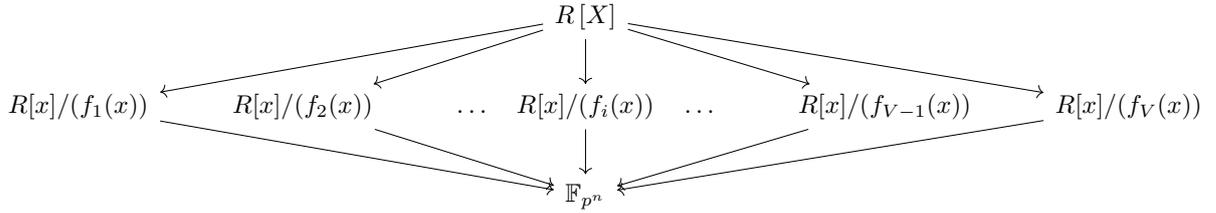
Figure 1.7: General diagram for FFS, NFS and variants.

Indeed, we know that every nonzero ideal can be written uniquely as a product of prime ideals. Moreover, because testing the B-smoothness of the norms of the prime ideals is computationally easier than directly testing the smoothness of the ideals, we compute two norms N_i and N_j of ϕ mapped to $R[x]/(f_i(x))$ and $R[x]/(f_j(x))$. A relation is then obtained when the latter are both B-smooth, for a smoothness bound B fixed during the complexity analysis.

For function fields: In the case of function fields, the elements of the factor basis also correspond to prime ideals. Testing the B-smoothness is done by computing the norm of these ideals and factoring the resulting univariate polynomial for which we know there exists a unique decomposition into a product of monic irreducible polynomials.

Each relation is therefore given by a polynomial ϕ for which the diagram gives a linear equation between the (virtual) logarithms of ideals of small norms coming from two distinct number or function fields. We omit details about the notion of virtual logarithms at this stage of the presentation and refer readers to [Sch05] and Chapter 4. For FFS, similar technicalities can be dealt with.

- Linear algebra: The relations obtained in the previous step form a system of linear equations where the unknowns are logarithms of ideals. This system is sparse with at most $O(\log p^n)$ non-zero entries per row, and can be solved in quasi-quadratic time using the block Wiedemann algorithm [Cop94].

- Individual logarithms: The previous step outputs the logarithms of ideals with norms smaller than the smoothness bound B used during sieving. The notion of *logarithm of an ideal* is of course not well-defined and results in the introduction of virtual logarithms mentioned above and detailed in Chapter 4. The goal of the algorithm is to compute the discrete logarithm of an arbitrary element in the target field. The commonly used approach for this step proceeds in two sub-steps. First, the target is subject to a smoothing procedure. The latter is randomized until after being lifted in one of the fields it becomes smooth (for a smoothness bound much larger than the bound B). Second, a special-q descent method is applied to each factor obtained after smoothing which is larger than the bound B. This allows to recursively rewrite their logarithms in terms of logarithms of smaller ideals. This is done until all the factors are below B, so that their logarithms are known. This forms what is called a descent tree where the root is an ideal coming from the smoothing step, and the nodes are ideals that get smaller and smaller as they go deeper. The leaves are the ideals just below B. We refer to [JLSV06, FGHT17] for details.

1.3.2 Description of the variants

Let us now describe the variants of NFS and see how they can be instantiated in our general setting.

Number Field Sieve. In this thesis, we call NFS, the simplest variant, where the ring R is $\mathbb{Z}$, there are only $V = 2$ number fields, and the polynomials f_1 and f_2 are constructed without using any specific form for p or the possible compositeness of n.

Multiple Number Field Sieve. The variant **MNFS** uses V number fields, where V grows to infinity with the size of the finite field. From two polynomials f_1 and f_2 constructed as in NFS, the $V - 2$ other polynomials are built as linear combinations of f_1 and f_2: we set $f_i = \alpha_i f_1 + \beta_i f_2$, for $i \geq 3$, where

the coefficients α_i, β_i are in $O(\sqrt{V})$. These polynomials have degree $\max(\deg(f_1), \deg(f_2))$ and their coefficients are of size $O(\sqrt{V} \max(||f_1||_\infty, ||f_2||_\infty))$.

There exist two variants of MNFS: an asymmetric one, coming from factoring [Cop93], where the relations always involve the first number field, and a symmetric one [BP14], where a relation can involve any two number fields. The asymmetric variant is more natural when one of the polynomials has smaller degree or coefficients than the others. When all the polynomials have similar features, at first it could seem that the symmetric case is more advantageous, since the number of possible relations grows as V^2 instead of V. However, the search time is also increased, since for each candidate ϕ, we always have to test V norms for smoothness, while in the asymmetric setup when the first norm is not smooth we do not test the others.

> **Question 15.** *What motivates the use of this variant of NFS?*

A relation will always involve two number fields and since MNFS considers V of them, this variant is intrinsically increasing the chances of finding smooth norms and thus speeding up the relation collection step.

(Extended) Tower Number Field Sieve. The **T**NFS or **exT**NFS variants cover the cases where $R = \mathbb{Z}[\iota]/h(\iota)$, where h is a monic irreducible polynomial. In the TNFS case, the degree of h is taken to be exactly equal to n, while the exTNFS notation refers to the case where $n = \kappa\eta$ is composite and the degree of h is η. Both TNFS and exTNFS can use either two number fields or $V \gg 2$ number fields. In the latter case, the prefix letter M is added refering to the MNFS variant. Details about (M)(ex)TNFS and their variants are given in [BGK15, KB16, KJ17, SS19]. A specific attention will be given to exTNFS in Chapter 4 as we provide a first implementation of this algorithm and a first record computation with it.

Remark 3. *In the literature, the denomination TNFS is sometimes used to describe a more general family of algorithms in which exTNFS is a special case. In this thesis, we will talk about TNFS when the degree of h is strictly greater than 1. This covers cases where n is either prime or composite. The variant exTNFS is only used when n is composite. However, the optimal complexities of exTNFS are only achieved when the factors of n are correctly balanced.*

> **Question 16.** *What motivates the use of this variant of NFS?*

When n is composite, the complexity of a computation done in medium characteristic with exTNFS can be viewed similarly as the complexity of a computation done with NFS at the boundary case between medium and large characteristic, meaning with a smaller constant c in the L_{p^n}-notation. This will be further explained in Chapter 4.

Special Number Field Sieve. The **S**NFS variant [JP14b] applies when the characteristic p is the evaluation of a polynomial of small degree with constant coefficients, which is a feature of several pairing construction families. Thus, the algorithm differs from NFS in the choice of the polynomials f_1 and f_2. To date, there is no known way to combine this efficiently with the multiple variant of NFS. However, it can be applied in the (ex)TNFS setup, giving STNFS and SexTNFS.

> **Question 17.** *What motivates the use of this variant of NFS?*

In this variant, the characteristic p is defined in a special way which leads to a sparse representation. This sparsity allows to select polynomials f_1, f_2 of small degrees and such that the product of the size of the coefficients is also very small. This will lead to smaller norms and thus a higher probability of finding relations.

Function Field Sieve. The FFS algorithm can be viewed in our general setting by choosing the polynomial ring $R = \mathbb{F}_p[\iota]$. The polynomials f_1 and f_2 are then bivariate, and therefore define plane

curves. The algebraic structures replacing number fields are then function fields of these curves. FFS cannot be combined efficiently with a multiple variant. In fact, FFS itself is already quite similar to a special variant; this explains this difficulty to combine it with the multiple variant, and to design an even more special variant. The tower variant is relevant when n is composite, and it can be reduced to a change of base field. We propose a new method to do so in Chapter 3.

In [JL06], Joux and Lercier proposed a slightly different setting. Although not faster than the classical FFS in small characteristic, it is much simpler, and furthermore, it gave rise to the pinpointing technique [Jou13] which is highly relevant when the characteristic is not so small. We recall their variant in Chapter 3 as we study its complexity at the boundary case $p = L_{p^n}(1/3)$. Several improvements to FFS exist when the finite field is a Kummer extension. This is not addressed in this thesis.

> **Question 18.** *What differences exist between the Function Field Sieve and Number Field Sieve algorithms?*

The FFS and NFS algorithms are very similar as the arithmetic of number fields and function fields are alike. However, noticeable differences are worth mentioning. These differences affect the overall complexity of the algorithms and allows FFS to achieve a lower complexity than NFS for a certain range of finite fields.

The first difference comes in the polynomial selection. As mentioned above, it is similar to SNFS even though FFS can be applied to any finite field of small characteristic which leads to an $L_{p^n}(1/3, (c/9)^{1/3})$ complexity with constant $c = 32$. Furthermore, Joux and Lercier introduced a method that appropriately selects polynomials for FFS. This method allows to use the pinpointing technique which we explain in Chapter 3, Section 3.2.2 as well as how it affects the complexity of FFS.

Other differences come from the computational cost of algorithms used in both contexts. One major difference is the cost of testing for smoothness. In the NFS algorithm, the smoothness of elements is tested using the Elliptic Curve Method (ECM) whose complexity is in $L_B(1/2, 1)$ where B is the smoothness bound. In order to amortize its cost for practical computations, it is run on a small set of elements (see Chapter 4 for more details).

On the other hand, testing for smoothness in FFS can be done in polynomial time. Indeed, there exist randomized algorithms that factor univariate polynomials over finite fields in polynomial time (for example the Cantor-Zassenhaus algorithm). There even exist deterministic algorithms (Shoup's algorithm) but only with polynomial time complexity in the average case (the polynomial time complexity in the worst-case for deterministic algorithms is still an open question).

Another noticeable difference is the absence of Schirokauer maps in the context of FFS. As we have not introduced Schirokauer maps yet, we refer to [Bar13, Section 6.6.3] for a detailed explanation as to how Schirokauer maps are replaced in FFS by valuations at infinity.

> **Question 19.** *What record computations have been done recently?*

Several of these algorithms have been implemented in order to perform record computations and estimate which sizes of finite fields are too small for security purposes. Among those algorithms, we have the Number Field Sieve, the Function Field Sieve, the Quasi-Polynomial algorithms and older $L(1/2)$-algorithms.

Not many of these algorithms have an open source implementation. It is the case for the Number Field Sieve algorithm implemented in CADO-NFS [cad]. The latter provides a complete implementation of NFS in C/C++. Recently, code for the TNFS variant has been added to it, see Chapter 5. All other implementations used for these records either partially use CADO-NFS for parts of the computation or remain unfortunately publicly unavailable.

We describe the most recent record computations (since 2016) in the following Figure 1.8. We attempt to classify these records using the three families of finite fields described by the L_{p^n}-notation. Since the latter is asymptotic, its translation for finite fields of fixed dimension is ambiguous and we will use the approximation $\alpha \approx \frac{\log \log p}{\log \log p^n}$. Let us illustrate this with two examples.

Example 1 (A record computation in medium characteristic)**.** *The recent record [MSST20] was done using the Function Field Algorithms over a 1051-bit field with a 22-bit characteristic p. From the definition*

of the L_{p^n} notation, we know that $\log p \approx (\log p^n)^\alpha$. We thus approximate

$$\alpha \approx \frac{\log(\text{bitsize of } p)}{\log(\text{bitsize of } p^n)} \approx \frac{\log 22}{\log 1051} \approx 0.44 \in (1/3, 2/3).$$

The finite field considered thus corresponds to a finite field that would be in the medium characteristic family.

Example 2 (A record computation in large characteristic). *Similarly, let us look at our own TNFS record computation in $\mathbb{F}_{p^6}$ of size 521 bit with characteristic p of 87 bits. Then*

$$\alpha \approx \frac{\log(\text{bitsize of } p)}{\log(\text{bitsize of } p^n)} \approx \frac{\log 87}{\log 521} \approx 0.71 \in (2/3, 1).$$

The finite field considered thus corresponds to a finite field that would be in the large characteristic family, however close to the second boundary.

$\mathbb{F}_{p^6}$	521	TNFS	2021 Chap 4
$\mathbb{F}_{p^6}$	423	NFS-HD	2020 [MR21]
$\mathbb{F}_p$	795	NFS	2020 [BGG⁺20]
$\mathbb{F}_{p^6}$	422	NFS-HD	2016 [GGMT17]
$\mathbb{F}_{p^5}$	324	NFS-HD	2016 [GGM17]
$\mathbb{F}_p$	1024	SNFS	2016 [FGHT17]
$\mathbb{F}_{p^3}$	593	NFS	2016 [Gré16]
$\mathbb{F}_p$	768	NFS	2016 [KDL⁺17]
$\mathbb{F}_{p^3}$	508	NFS	2016 [GMT16]

$\mathbb{F}_{2^{30750}}$	30750	QP	2019 [GKL⁺20]
$\mathbb{F}_{3^{3054}}$	4841	QP	2016 [ACMCC⁺18]

$\mathbb{F}_{p^{50}}$	1051	FFS	2020 [MSST20]

0     $\frac{1}{3}$ $\frac{2}{3}$ 1 α

Small char   **Medium char**   **Large char**
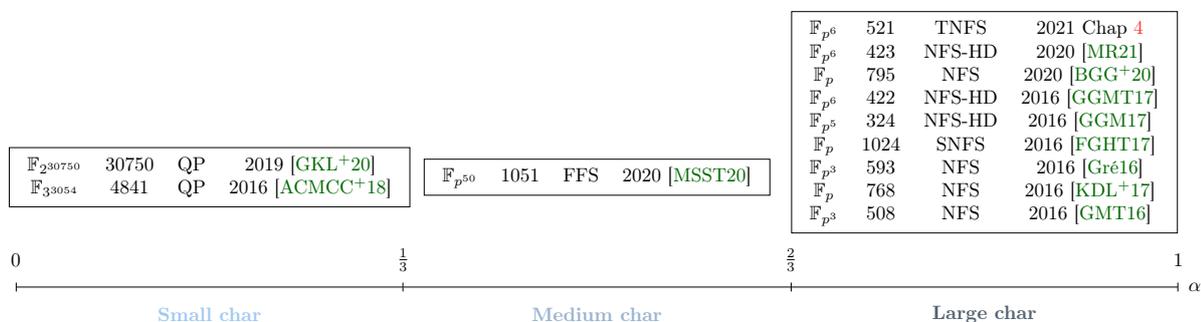
Figure 1.8: Latest record computations (since 2016) for discrete logarithm computations in $\mathbb{F}_{p^n}$ in each family of finite fields. We give: finite field, size of p^n, algorithm used, year/reference (if any).

A complete list of record computations is given in [Gré16].

Question 20. *Which other variants of NFS have been implemented?*

As we can see from Figure 1.8, the record computations done in large characteristics mostly use the Number Field Sieve algorithm. However, the latter has many variants that asymptotically improve the complexity. Among these variants, only TNFS was implemented and used for a record, see Chapters 4 and 5. Other variants such has MNFS have not been implemented yet. Many obstacles come along the implementation of a multiple number field variant such as the choice of the number of number fields V, the correct polynomials to select and the adaptation of the code to its specific context. The (non-asymptotic) gain of MNFS for record computations is also not entirely clear. Therefore, MNFS still awaits its first record which constitutes an interesting topic of research.

The special variant of NFS only differs in the choice of polynomials and thus record computations have been done with this variant using the CADO-NFS software. Similarly NFS-HD refers to the NFS algorithm where the dimension of the sieving space is greater than 2 and the latter was also implemented in CADO-NFS (no longer maintained).

Question 21. *What key and group sizes are recommended?*

The US *National Institute of Standards and Technology* (NIST) and the French *National Agency for the Security of Information Systems* (ANSSI) are two national institutes that provide standards for cryptographic protocols. In particular, they give recommendations for key sizes and group sizes of all cryptographic protocols that are commonly used. We summarize in Table 1.2 their recommendations for protocols based on the hardness of the discrete logarithm problem.

Agency	Date	Size of group	Size of key
NIST	2019-2030	2048	224
	> 2030	3072	256
ANSSI	2021-2030	2048	200
	> 2030	3072	200

Table 1.2: Recommendations from NIST and ANSSI for discrete logarithm key sizes and groups. All sizes are given in bits.

One can note that these recommendations do not take into account specificities of the finite field considered. Indeed, we have seen throughout this chapter that the hardness of solving the discrete logarithm problem is very dependent on the characteristics of the group considered. More precisely for finite fields, it depends on the relation between the characteristic p and the extension degree n, the compositeness of n and special properties of p (more specifically, its sparsity).

Figure 1.8 illustrates these differences. Indeed, for finite fields of small characteristic, record computations were performed on finite fields whose size exceeds the recommended size (eg, $30750 \gg 2048$, even 3072). In large characteristic, the largest finite field for which a record computation has been performed is of size 1024 bit, but the record was performed with SNFS which can be used when the characteristic p is of a special form.

Hence, it is important to keep in mind that in order to guarantee a sufficient level of security, the size of the finite field must be chosen in line with the specificities of p and n. In practice, fields of small characteristic are avoided, and for practical reasons prime fields are often chosen.

Finally, key sizes correspond to the size of the subgroup in which we consider DLP and are usually chosen to be twice the level of security wanted to avoid attacks coming from Pollard's kangaroo algorithm for example.

Chapter 2

Lattices and related hard computational problems

In this chapter, we introduce some fundamental notions in the theory of lattices specific to their use for cryptanalysis. We restrict the presentation of lattices to notions that are useful in this thesis.

In Section 2.1, we recall general definitions and results proper to lattices and related algorithmic problems. We focus on two fundamental computationally hard lattice problems: the *Shortest Vector Problem* (SVP) and the *Closest Vector Problem* (CVP). The hardness of SVP in particular is at the center of the security estimates of lattice-based cryptosystems.

We then cover algorithms that give both exact and approximate solutions to these problems in Section 2.2. More precisely, we first focus on enumeration algorithms which are the oldest examples of algorithms that provide exact solutions to SVP and CVP. We focus on this family of algorithms as lattice enumeration will also be used in this thesis in the context of discrete logarithm computations to collect algebraic relations, see Chapter 4.

Finally, we look at the main lattice basis reduction algorithms which provide approximate solutions to SVP and CVP in Section 2.3. These reduction algorithms such as LLL and BKZ are extensively used for cryptanalysis and also became important tools in algebraic number theory. In this thesis, we will use them specifically in the context of key recovery, see Part III.

Contents

2.1 Lattices

2.1.1 Euclidean lattices

We start by recalling a few notations and definitions. Vectors are written in **bold**. The operators $||\cdot||$ and $\langle \cdot, \cdot \rangle$ denote the Euclidean norm and the inner product in the Euclidean space $\mathbb{R}^n$. Definitions and results

in this chapter can be adapted to any norm but we focus on the ℓ_2 norm as it is the most commonly used in cryptography.

Definition 9 (Lattice). *Let $B = \{\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_k\} \subset \mathbb{R}^{n \times k}$ be a set of $k \leq n$ linearly independent vectors. The lattice generated by B is the set $\mathcal{L}(B)$ defined by*

$$\mathcal{L}(B) \;:=\; \left\{ \sum_{i=1}^{k} x_i \boldsymbol{b}_i : x_1, x_2, \ldots, x_k \in \mathbb{Z} \right\}.$$

Lattices are discrete additive subgroups of $\mathbb{R}^n$. The notion of "discrete" refers to the fact that there exists a real number $\lambda > 0$ such that any two points in $\mathcal{L}$ are distanced by at least this quantity λ. We will discuss this λ value more extensively below.

The set B in the above definition is called a basis of $\mathcal{L}(B)$. The basis of a lattice is clearly non-unique. In fact, many algorithmic problems related to lattices reduce to finding a "good" basis of the lattice for a notion of "good" which we will define later. We will simply write $\mathcal{L}$ for $\mathcal{L}(B)$ when the context is not ambiguous. The dimension of the lattice is n and its rank is k. The lattice is said to be full-rank if $k = n$. An example of a full-rank 3-dimensional lattice and its basis $B = \{\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3\}$ is given in Figure 2.1.

Lattices differ from the notion of vector space by the fact that only integer coefficients are allowed in the linear combinations. However, many similar mathematical quantities exist between vector spaces and lattices.

Let us start with the notion of determinant. Let $\mathcal{P}(B)$ be the fundamental parallelepiped of the basis $B = \{\mathbf{b}_1, \mathbf{b}_2, \cdots, \mathbf{b}_k\}$, that is $\mathcal{P}(B) = \{\sum_i x_i \mathbf{b}_i : 0 \leq x_i < 1\}$. Then the determinant of a lattice $\mathcal{L}(B)$ is the volume of $\mathcal{P}(B)$. More precisely, we have the following definition.

Definition 10 (Determinant). *The determinant of a lattice $\mathcal{L}$ with basis B is the volume of the fundamental parallelepiped,* i.e.,

$$\det(\mathcal{L}) = \sqrt{\det(B^T B)}.$$

If $\mathcal{L}$ is full-rank, then the determinant is given by $\det(\mathcal{L}) \;=\; |\det([\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_n])|$.
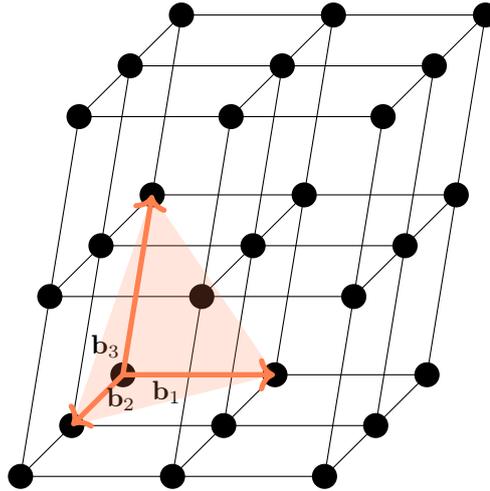
Figure 2.1: Lattice in dimension 3 with a basis $B = \{\mathbf{b}_1, \mathbf{b}_2, \mathbf{b}_3\}$ and its fundamental parallelepiped.

Remark 4. *The determinant of a lattice $\mathcal{L}$ is independent from the choice of the basis B.*

The determinant can be upper-bounded by the following inequality.

Theorem 3 (Hadamard Inequality). *For a given lattice $\mathcal{L} \subset \mathbb{R}^n$ with basis $B = \{\boldsymbol{b}_1, \boldsymbol{b}_2, \cdots, \boldsymbol{b}_k\}$, we have $\det(\mathcal{L}) \leq \prod_i ||\boldsymbol{b}_i||$.*

More importantly, the determinant can be computed in polynomial time. Recall that the Gram-Schmidt orthogonalization (GSO) process transforms the basis $B = \{\mathbf{b}_1, \cdots, \mathbf{b}_k\}$ into a basis of orthogonal vectors $B^* = \{\mathbf{b}_1^*, \cdots, \mathbf{b}_k^*\}$ that span the same space as B. The basis B^* is defined as follows: let $\mathbf{b}_1^* := \mathbf{b}_1$ and, for $2 \leq i \leq k$, let $\mathbf{b}_i^* := \mathbf{b}_i - \sum_{j=1}^{i-1} \mu_{i,j} \mathbf{b}_j^*$, where, for $1 \leq j < i \leq k$, we have $\mu_{i,j} := \frac{\langle \mathbf{b}_i, \mathbf{b}_j^* \rangle}{\langle \mathbf{b}_j^*, \mathbf{b}_j^* \rangle}$.

The Gram-Schmidt orthogonalization of a basis can be computed in polynomial time and from the Gram-Schmidt orthogonalization of B, we also have

$$\det(\mathcal{L}(B)) = vol(\mathcal{P}(B)) = \prod_{i=1}^{k} ||\mathbf{b}_i^*||.$$

Another fundamental notion related to lattices is the minimum distance λ between any two vector points in the lattice.

Definition 11 (Minimum distance). *Let $\mathcal{L} \subset \mathbb{R}^n$ be a lattice. The minimum distance of $\mathcal{L}$ is defined as*

$$\lambda(\mathcal{L}) = \inf\{||\boldsymbol{x} - \boldsymbol{y}|| : \boldsymbol{x}, \boldsymbol{y} \in \mathcal{L}, \boldsymbol{x} \neq \boldsymbol{y}\}$$

or equivalently

$$\lambda(\mathcal{L}) = \inf\{||\boldsymbol{v}|| : \boldsymbol{v} \in \mathcal{L} \setminus \{\boldsymbol{0}\}\}.$$

This minimum distance corresponds to the length of a shortest non-zero lattice vector. The equivalence in the definition comes from the fact that lattices are closed under addition and subtraction. Hence, $\mathbf{v} = \mathbf{x} - \mathbf{y} \in \mathcal{L} \setminus \{\mathbf{0}\}$. Moreover, this definition also implies that $\lambda(\mathcal{L})$ is the smallest radius r such that a ball of radius r contains at least one non-zero lattice vector point.

Theorem 4. *There is a lattice vector $\boldsymbol{v} \in \mathcal{L}$ of length exactly $\lambda(\mathcal{L})$.*

The proof of Theorem 4 given in [MG02, Section 1.2] relies on the following result.

Theorem 5. *Let $\mathcal{L} \subset \mathbb{R}^n$ be a lattice with basis $B = \{\boldsymbol{b}_1, \boldsymbol{b}_2, \cdots, \boldsymbol{b}_k\}$ and $B^* = \{\boldsymbol{b}_1^*, \boldsymbol{b}_2^*, \cdots, \boldsymbol{b}_k^*\}$ the Gram-Schmidt orthogonalization of B. Then*

$$\lambda(\mathcal{L}) \geq \min_i ||\boldsymbol{b}_i^*|| > 0.$$

We refer to [MG02, Theorem 1.1] for the proof of this lower-bound.

The notion of length of the shortest non-zero vector can easily be generalized with the following definition.

Definition 12 (Successive minima). *Consider a lattice $\mathcal{L} \subset \mathbb{R}^n$ of rank k. For $i \in [1, k]$, the ith successive minimum is defined as the quantity*

$$\lambda_i(\mathcal{L}) = \inf\{r : \dim(\text{span}(\mathcal{L} \cap B_r(0))) \geq i\},$$

where $B_r(0) = \{\boldsymbol{x} \in \mathbb{R}^n : ||\boldsymbol{x}|| \leq r\}$.

This definition provides the following sequence of parameters $\lambda(\mathcal{L}) = \lambda_1(\mathcal{L}) \leq \lambda_2(\mathcal{L}) \leq \cdots \leq \lambda_k(\mathcal{L})$ which gives the successive minima of the lattice $\mathcal{L}$.

On the contrary of the determinant which can be computed in polynomial time, finding a shortest non-zero vector in a lattice is hard. In the late 19th century, Minkowski provided an upper bound on the length of the shortest vector in a lattice. We first state his convex body theorem on which relies his upper bound.

Theorem 6 (Minkowski's convex body theorem). *Let $\mathcal{L}$ be an n-dimensional full-rank lattice and $S \subset \mathbb{R}^n$ a symmetric convex body such that $\mathrm{vol}(S) > 2^n \det(\mathcal{L})$. Then S contains a non-zero lattice point.*

The above theorem is then used to provide the following upper bound.

Corollary 2 (Minkowski's bound). *Given an n-dimensional full-rank lattice $\mathcal{L}$, there exists a non-zero vector $\boldsymbol{v} \in \mathcal{L}$ such that $||\boldsymbol{v}|| \leq \sqrt{n} \det(\mathcal{L})^{1/n}$. In other words, $\lambda(\mathcal{L}) \leq \sqrt{n} \det(\mathcal{L})^{1/n}$.*

We refer to [MG02, Section 1.3] for the proofs of these results.

Despite Minkowski's upper bound, it can be noted that $\lambda(\mathcal{L})$ is usually smaller than $\sqrt{n} \det(\mathcal{L})^{1/n}$. Moreover, his proof does not provide any computational method to efficiently find these short vectors. Finding vectors of length $\lambda(\mathcal{L})$ is among the hard algorithmic problems related to lattices which we will now discuss.

2.1.2 Algorithmic problems related to Euclidean lattices

Lattices have become tools from which computationally hard related problems are used to design secure cryptographic primitives. As mentioned above, finding the minimum distance in a lattice is an example of a hard problem for which no efficient algorithm is known. We will now discuss two of the fundamental hard problems related to lattices: the *Shortest Vector Problem* and the *Closest Vector Problem*, commonly known as SVP and CVP.

Definition 13 (Shortest Vector Problem (SVP)). *For a given lattice $\mathcal{L} \subset \mathbb{R}^n$ with basis B, find a shortest non-zero vector $\boldsymbol{v} \in \mathcal{L}(B)$ with $||\boldsymbol{v}|| = \lambda(\mathcal{L}(B))$.*

Whether SVP is an NP-hard problem has long remained an open question. In 1981, van Emde Boas [vEB81] proved that SVP is NP-hard for the ℓ_∞ norm and conjectured the NP-hardness for the ℓ_p norm for $p \geq 1$. Finally, in 1996 Ajtai [Ajt98] proved the NP-hardness of SVP for the ℓ_p norm for $p \geq 1$ but only under randomized reductions, meaning that an algorithm for SVP would give a randomised algorithm for any problem in NP. The NP-hardness of SVP under deterministic reductions for ℓ_p norms is still an open question.

A generalization of SVP is the Closest Vector Problem. For $\mathbf{t} \in \mathbb{R}^n$, the distance between the vector $\mathbf{t}$ and the lattice $\mathcal{L}$ is denoted $\mathrm{dist}(\mathbf{t}, \mathcal{L})$ and represents $\mathrm{dist}(\mathbf{t}, \mathcal{L}) = \min_{\mathbf{v} \in \mathcal{L}}(||\mathbf{v} - \mathbf{t}||)$.

Definition 14 (Closest Vector Problem (CVP)). *For a given lattice $\mathcal{L} \subset \mathbb{R}^n$ with basis B and a target element $\boldsymbol{t} \in \mathbb{R}^n$, find a lattice vector $\boldsymbol{v} \in \mathcal{L}$ closest to the target element $\boldsymbol{t}$, i.e., $||\boldsymbol{v} - \boldsymbol{t}|| \leq \mathrm{dist}(\boldsymbol{t}, \mathcal{L})$.*

If the target element $\mathbf{t}$ is the origin, then we are looking for a shortest vector. CVP was proven to be NP-hard with the ℓ_p norm for $p \geq 1$ and ℓ_∞ in 1981 by van Emde Boas [vEB81]. We refer to [MG02, Chapter 3] for a proof that CVP is NP-hard by reduction from the subset sum problem.

Both SVP and CVP are illustrated in Figure 2.2.
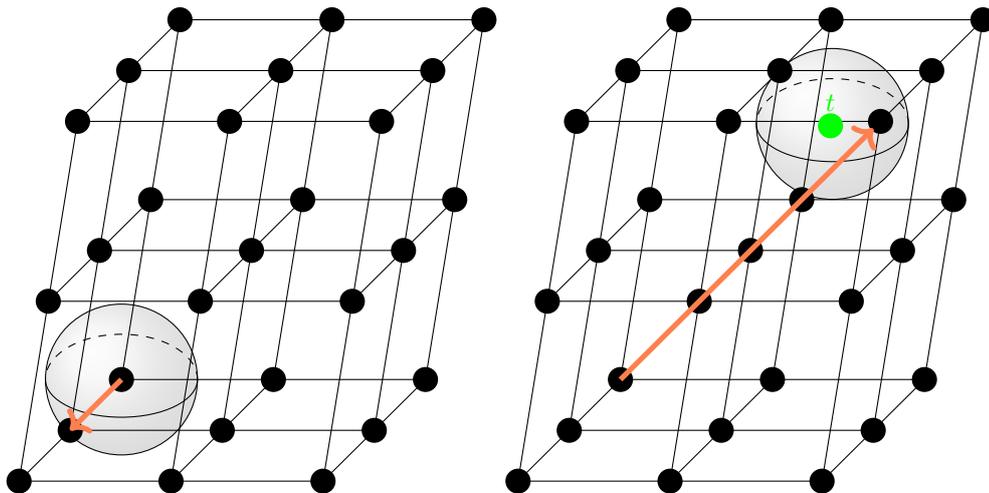
Figure 2.2: Finding a shortest non-zero vector (left) and a closest non-zero vector (right) to the target t in a 3-dimensional lattice.

Question 22. *What algorithms can one use to solve SVP and CVP?*

Both SVP and CVP are hard problems to solve for general lattices and no known efficient algorithms exist as of today. The hardness of these problems are of course dependent on the dimension n of the lattice considered. In dimension 2, SVP can be solved using the Lagrange-Gauss algorithm in polynomial time (1801). However, when the lattice dimension is larger, which is often the case in cryptography, the best known algorithms to solve SVP and CVP are not in polynomial time.

There exists three main families of algorithms to solve SVP: enumeration algorithms, sieving algorithms and Voronoi cell based algorithms.

The family of enumeration algorithms, originally from Pohst [Poh81] and Kannan [Kan83], provide the oldest examples of deterministic algorithms that lead to solutions of SVP, however at the cost of super-exponential running time, *i.e.,* more than 2^n, and in this case, at best $2^{n \log n}$. These algorithms are interesting as they perform well in practice for relatively small dimensions. We discuss enumeration algorithms in more details in Section 2.2.

Sieving algorithms are probabilistic algorithms. There exists several variants of sieve algorithms but all proceed with the following generic idea: from a given list of vectors in the lattice, the algorithm looks at pairs $\mathbf{v}, \mathbf{w} \in \mathcal{L}$ such that $||\mathbf{v} - \mathbf{w}|| < ||\mathbf{v}||$ and if so, replaces the vector $\mathbf{v}$ by the new, shorter vector $\mathbf{v} - \mathbf{w}$. Depending on the sieve, another vector might get replaced, for example the currently longest in the database. They offer an asymptotically faster alternative to enumeration with single exponential running time, however at the cost of randomization and exponential space complexity. We mention the first sieve algorithm due to Ajtai, Kumar and Sivakumar [AKS01], known as the AKS algorithm, which was the first algorithm to ever solve a hard lattice problem in single exponential time 2^{cn}, where $c > 0$ is a constant. Later variants improved the original AKS algorithm by lowering the constant c. A provable variant, `ListSieve` was introduced by Micciancio and Voulgaris in 2010 [MV10c] along with a heuristic, more practical variant `GaussSieve`. More heuristic sieve algorithms exist, each improving on the constant c [NV08, PS09, WLTB11, ZPH14, BGJ15]. The fastest variant as of today is given in [BDGL16, MLB17] where $c = 0.292$. Similar algorithms exist also for particular lattices such as ideal lattices [Sch13, BL16].

Finally, we can also mention the Voronoi cell computation approach [MV10b] which provides currently the asymptotically fastest known deterministic algorithms with time complexity 2^{2n} and space complexity 2^n. These algorithms rely on the computation of the Voronoi cell of the input lattice. The latter are however not competitive in practice. We refer to [HPS11a] for a description and comparison of these three families of SVP solvers.

Enumeration and Voronoi cell algorithm can usually be adapted to solve CVP as well with similar

cost. On the other hand, it is less obvious whether lattice sieving methods can solve CVP while keeping the same cost as for solving SVP. In [Laa16], the author presents two different approaches for solving CVP with sieving methods.

Because no efficient algorithm exists to solve SVP and CVP when the dimension is greater than 2, approximate versions of these problems exist. These versions allow for an approximation factor $\gamma \geq 1$. We denote these variants γ-SVP and γ-CVP. The exact version is recovered if one sets $\gamma = 1$.

Definition 15 (Approximate Shortest Vector Problem (γ-SVP))**.** *For a given lattice $\mathcal{L} \subset \mathbb{R}^n$ with basis B and for any $\gamma \geq 1$, find a short non-zero vector $\boldsymbol{v} \in \mathcal{L}(B)$ with $||\boldsymbol{v}|| \leq \gamma \cdot \lambda(\mathcal{L}(B))$.*

A variant of SVP that has been particularly relevant in cryptography is the unique shortest vector problem (uSVP) with its approximate version γ-uSVP which corresponds to the problem of finding the shortest non-zero vector of the lattice, however with the guarantee that the shortest vector is at least γ times smaller than the next (non-parallel) shortest lattice vector.

Definition 16 (Approximate Closest Vector Problem (γ-CVP))**.** *For a given lattice $\mathcal{L} \subset \mathbb{R}^n$ with basis B, a target element $\boldsymbol{t} \in \mathbb{R}^n$ and for any $\gamma \geq 1$, find a lattice vector $\boldsymbol{v} \in \mathcal{L}$ close to the target element $\boldsymbol{t}$, i.e., $||\boldsymbol{v} - \boldsymbol{t}|| \leq \gamma \cdot dist(\boldsymbol{t}, \mathcal{L})$.*

These approximate versions of SVP and CVP are illustrated in Figure 2.3. Naturally, the hardness of γ-SVP and γ-CVP decreases when the approximation factor γ increases.
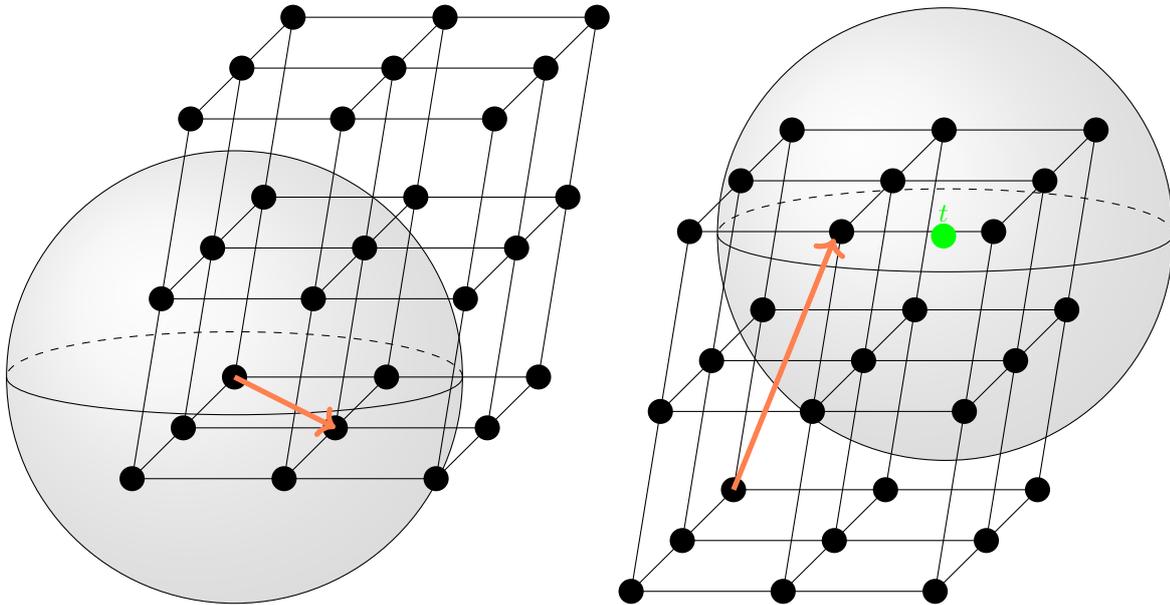
Figure 2.3: Solving γ-SVP (left) and γ-CVP (right) with $\gamma = 3$.

Question 23. *What are the best algorithms to solve γ-SVP and γ-CVP?*

There is an obvious trade-off between the running-time and the approximation factor of algorithms that solve γ-SVP and γ-CVP. Indeed, there exist polynomial time algorithms for these approximate variants but only for approximation factors large enough. Efficient algorithms that solve γ-SVP rely on lattice basis reduction techniques. These algorithms run in polynomial time for approximation factors γ as small as $2^{O(n \log \log n / \log n)}$, where n is the dimension of the lattice. If the approximation factor is

asymptotically smaller, then the algorithms have exponential running time in the lattice input size. We discuss reduction algorithms in Section 2.3.

As for γ-CVP, there are two main approaches. First, Babai's nearest plane algorithm [Bab85] developed by Babai in 1986 runs in polynomial time in the input size for $\gamma = 2(2/\sqrt{3})^n$ where n is the dimension of the lattice. Second, algorithms that solve γ-SVP such as reduction algorithms can usually be modified to solve γ-CVP. Note that the AKS sieving algorithm was also adapted to solve γ-CVP [AKS02]. In general, the most efficient algorithms that solve γ-CVP are asymptotically as fast as those that solve γ-SVP.

> **Question 24.** *What relations are there between γ-SVP and γ-CVP?*

The Shortest and Closest Vector Problems are very similar. It is natural to wonder whether there exists any reductions between these problems.

Kannan embedding technique. One way to solve γ-CVP is to solve γ-SVP in an adequately constructed lattice. Let $\mathcal{L}$ be an n-dimensional lattice with basis $B = \{\mathbf{b}_1, \mathbf{b}_2, \cdots, \mathbf{b}_n\}$ and let $\mathbf{t} \in \mathbb{R}^n$ be the target vector in CVP. A solution to CVP is then a vector $\mathbf{v} = \sum_{i=1}^n x_i \mathbf{b}_i \in \mathcal{L}$, $x_i \in \mathbb{Z}$, such that $||\mathbf{v} - \mathbf{t}||$ is small. Let $\mathbf{w} = \mathbf{v} - \mathbf{t}$. The idea behind the embedding technique is to construct a lattice $\mathcal{L}'$ such that the vector $\mathbf{w}$ is contained in $\mathcal{L}'$. By solving SVP in $\mathcal{L}'$ and finding $\mathbf{w}$ one can then recover $\mathbf{v}$. The basis B' of $\mathcal{L}'$ is one dimension higher and consists of the vectors $B' = \{(\mathbf{b}_1, 0), (\mathbf{b}_2, 0), \cdots, (\mathbf{b}_n, 0), (\mathbf{t}, C)\}$ for some positive constant C. It is easy to see that the vector $\mathbf{w}$ is found as $(x_1, x_2, \cdots, x_n, -1) \cdot B' = (\mathbf{v} - \mathbf{t}, C)$.

The main issue with this technique is the size of the vector $\mathbf{w}$. If the latter is much larger than the shortest vectors in $\mathcal{L}'$, then SVP will not find it.

One can then also wonder whether it is possible to reduce γ-SVP to γ-CVP. The first naive idea for a reduction would be to solve γ-SVP by solving γ-CVP with target vector $\mathbf{t} = \mathbf{0}$. Indeed, when looking for the shortest vector of a lattice $\mathcal{L}$, we are intrinsically looking for the closest vector to the origin. However, this naive reduction does not work as CVP will return the vector $\mathbf{0}$ which is not a solution of SVP. It was nonetheless proven in [GMSS99] that γ-SVP is Turing-reducible to γ-CVP in polynomial time for all $\gamma \geq 1$ meaning that given an oracle for γ-CVP, it is possible to compute answers to γ-SVP. The reduction preserves the dimension and the approximation factor γ.

2.1.3 Ideal and module lattices

In recent years, cryptographers have turned their attention to lattices with more algebraic structure. Indeed, while lattices provide a high level of security, protocols based on Euclidean lattices are usual quite inefficient due to the key size. Ideal and module lattices are examples of structured lattices used in lattice-based schemes to improve the efficiency of the protocols. We briefly introduce them here and refer to [SSTX09, PHS19, LPR10, LS15, LPSW19, KEF19] for examples of articles working with ideal and module lattices.

Let us first start with the notion of cyclic lattice, used for example in NTRUEncrypt [HPS98]. Reinvestigated in [Mic04], they refer to lattices that are invariant under a shift-rotation operator. In other words, we have the following definition.

> **Definition 17.** *A set $\mathcal{L} \subset \mathbb{R}^n$ is a cyclic lattice if $\mathcal{L}$ is an ideal in $\mathbb{Z}[x]/(x^n - 1)$.*

Ideal and module lattices are then generalization of cyclic lattices. An element in an ideal lattice corresponds to a polynomial: $(a_0, a_1, \cdots, a_{n-1}) \in \mathcal{L}$ is the polynomial $a = \sum_{i=0}^{n-1} a_i x^i$.

> **Definition 18** (Ideal lattice). *The lattice $\mathcal{L}(B) \subset \mathbb{R}^n$ is an ideal lattice if there exists a monic polynomial $g \in \mathbb{Z}[x]$ such that the lattice element $(a_0, a_1, \cdots, a_{\deg g-1}) \in \mathcal{L}$ if and only if $(a'_0, a'_1, \cdots, a'_{\deg g-1}) \in \mathcal{L}$ where $a' = xa \pmod{g}$.*

The following result describes the family of full-rank ideal lattices.

Lemma 1. *Let f be an irreducible monic integer polynomial of degree n. Then, every ideal I of $\mathbb{Z}[x]/(f)$ is isomorphic to a full-rank lattice in $\mathbb{Z}^n$.*

Module lattices are defined similarly as generalizations of ideal lattices. A basis of a module lattice is a block matrix where each block is the basis of an ideal lattice.

2.1.4 Random lattices and random bases

In order to correctly draw conclusions from either experiments or theoretical results, one must consider lattices that are random in a mathematical sense. For example, lattices used in cryptography or in algorithmic number theory are not necessarily random in the sense that the first minimum is often much shorter than all the other minima.

A random lattice is defined as a lattice chosen from the natural probability distribution on the set of all lattices. This distribution comes from the notion of Haar measures of classical groups, which we do not explain here. We refer to [NS06] for more details.

Any (random) lattice has an infinite amount of bases and thus comes the following problem of selecting a random one. There exists no definition of a random basis, but in simple terms, a random basis should not be reduced at all, meaning with particularly short vectors. More details are again given in [NS06].

In this thesis, when experimentally validating hypotheses, we use random integer-valued bases of size $k = n$ using two different constructions made explicit below.

The Ajtai-type bases. The first construction is inspired by the lattice bases used in [Ajt03], and also used in [NS06]. The Ajtai-type basis is generated as a lower triangular matrix. The diagonal values are defined as $A_i := \lfloor 2^{(2n-i+1)^{\alpha}} \rfloor$, where α is a parameter we vary. The other entries $a_{i,j}$ are random integers bounded by the integer on the diagonal in their row, i.e. $|a_{i,j}| \le A_i/2$, for all $1 \le j < i \le n$. That means, the basis is given by the rows of the following matrix:

$$\begin{pmatrix} A_1 & & & & \\ a_{2,1} & A_2 & & & \\ a_{3,1} & a_{3,2} & A_3 & & \\ \vdots & & & \ddots & \\ a_{n,1} & a_{n,2} & & a_{n,n-1} & A_n \end{pmatrix}.$$

The Goldstein-Mayer bases. This second construction comes from Goldstein-Mayer [GM03]. Let p be a prime number with $2^{\lfloor \beta n \rfloor}$ bits, for some parameter β we vary. Further let $a_1, a_2, \ldots, a_{n-1}$ be randomly generated numbers in $\{1, 2, \ldots, p\}$. The prime lattice basis is given by the rows of the following matrix:

$$\begin{pmatrix} p & & & & \\ a_1 & 1 & & & \\ a_2 & & 1 & & \\ \vdots & & & \ddots & \\ a_{n-1} & & & & 1 \end{pmatrix}.$$

A common result used in the analysis of lattice-based algorithms is the Gaussian Heuristic. For a random lattice, it stipulates that the number of lattice points inside a measurable set $\mathcal{S} \subset \mathbb{R}^n$ is proportional to the volume of $\mathcal{S}$. In other words,

Gaussian heuristic. *For a full-rank lattice $\mathcal{L} \subset \mathbb{R}^n$ and a set $\mathcal{S} \subset \mathbb{R}^n$, the number of points in $\mathcal{L} \cap \mathcal{S}$ is roughly the ratio of the volumes,* i.e., $vol(\mathcal{S})/vol(\mathcal{L})$.

2.2 Enumeration to solve exact SVP and CVP

The first and oldest family of algorithms that solve SVP and CVP are enumeration algorithms. Enumeration techniques rely on an exhaustive enumeration of all lattice vector points contained in a convex set. The first enumeration algorithm was proposed in 1981 by Pohst [Poh81] and a slightly different approach was given by Kannan [Kan83] in 1983. Generalizations of these algorithm were given by Fincke and Pohst [FP85] in 1985, and later in 1987 by Kannan [Kan87].

These algorithms differ in the choice of the chosen convex set: Pohst initially considered a hypersphere and Kannan a rectangular parallelepiped. Later on, Kannan chose to enumerate in a hyper-parallelepiped and Fincke and Pohst generalized Pohst's original algorithm to a hyper-ellipsoid contained in Kannan's hyper-parallelepiped. Since these algorithms are similar, they are often referred to as the Kannan-Fincke-Pohst (KFP) algorithm.

In 1994, Schnorr and Euchner [SE94] introduced a heuristically faster variant relying on slightly different techniques. Their algorithm is the most commonly used for cryptanalysis. Further work on enumeration algorithms continued with major practical improvements due to pruning techniques. The latter consists in reducing the search space by eliminating some branches of the enumeration tree built by the algorithms which are less likely to produce the shortest solution, see [GNR10]. New lattice enumeration algorithms were also introduced in [MW15] achieving both the optimal asymptotic complexity and maintaining practicability.

Question 25. *Why is enumeration interesting?*

The attractiveness of enumeration algorithms comes from both its practicality and theoretical performances. One particular advantage that enumeration algorithms have over sieving algorithms is a linear space complexity in the size of the input. Within the class of polynomial space algorithms, the enumeration algorithms are the fastest known algorithms to find exact solutions to SVP. Their worst-case complexity is $2^{O(n \log n)}$ for lattices of dimension n. Note that sieving algorithms will achieve an asymptotic complexity of $2^{O(n)}$ but at the cost of an $2^{O(n)}$ space requirement. In practice, they perform well up to moderately large dimensions, allowing these enumeration algorithms to be used to estimate security parameters for lattice-based schemes. Recent progress on sieving algorithms, both theoretical [HKL18] and practical [Duc18, LM18, ADH$^+$19], have however lowered the crossover point between sieving and enumeration. Enumeration algorithms outperform sieving algorithms up to dimension 70 in practice. In particular, for exact-SVP the General Sieve Kernel (known as G6K [ADH$^+$19]) outperforms pruned enumeration implemented in FPLLL by dimension 70.

2.2.1 General framework of enumeration algorithms

Any enumeration algorithm takes as input a lattice basis and outputs non-zero vectors of the lattice contained in a specific region. The output of the enumeration algorithm can then be used to find the shortest non-zero lattice vector *i.e.,* a solution to exact-SVP. The considered region is often an n-sphere.

In this section, we consider full-rank lattices of dimension n. Let $B = \{\mathbf{b}_1, \cdots, \mathbf{b}_n\}$ be a basis of the lattice $\mathcal{L}$ considered and let R be a bound chosen such that $R \geq ||\mathbf{b}_1||$ (in order to obtain at least one non-trivial solution). The algorithm will enumerate all vectors $\mathbf{v} = (v_1, v_2, \cdots, v_n) \in \mathbb{Z}^n$ such that the vectors $\mathbf{c} = \sum_{i=1}^n v_i \mathbf{b}_i \in \mathcal{L}$ have norm smaller than R.

Now, recall that from the Gram-Schmidt process, we have the following relation $\mathbf{b}_i = \mathbf{b}_i^* + \sum_{j=1}^{i-1} \mu_{i,j} \mathbf{b}_j^*$. One can replace the $\mathbf{b}_i$'s in the expression of $\mathbf{c}$ and get

$$\mathbf{c} = \sum_{i=1}^n \left(v_i \mathbf{b}_i^* + \sum_{j=1}^{i-1} v_i \mu_{i,j} \mathbf{b}_j^* \right) = \sum_{j=1}^n \left((v_j + \sum_{i=j+1}^n \mu_{i,j} v_i) \mathbf{b}_j^* \right).$$

The enumeration algorithm is thus looking for the linear combination vectors $(v_1, \cdots, v_n)$ such that

$$||\mathbf{c}||^2 = \sum_{j=1}^n \left((v_j + \sum_{i=j+1}^n \mu_{i,j} v_i)^2 ||\mathbf{b}_j^*||^2 \right) \leq R^2.$$

Question 26. *How to enumerate efficiently?*

The question now remains how to efficiently find these vectors $\mathbf{c}$ of bounded norm. The main idea of enumeration algorithms is to reduce the search of the entire n-space to an exhaustive search in an interval, hence a space of dimension 1. In other words, let $k < n$ and suppose the coefficients $v_{k+1}, \cdots, v_n$ of the vector $\mathbf{v}$ are fixed. Then, there exists only a finite number of admissible values for v_k which lie in a

bounded interval. These are the values that need to be enumerated.

2.2.2   Constructing an enumeration tree

Enumeration algorithms consider projections of the lattice $\mathcal{L}$. Since the norm of vectors cannot increase under orthogonal projections, enumeration algorithms proceed recursively by looking at the orthogonal projections π_k on the set $\{\mathbf{b}_1, \cdots, \mathbf{b}_{k-1}\}^\perp$ for decreasing values of k and π_1 is the identity. The projection of the vector $\mathbf{c}$ for a given $k = 1, 2, \cdots, n$ is then

$$\pi_k(\mathbf{c}) = \sum_{j=1}^{n} \left((v_j + \sum_{i=j+1}^{n} \mu_{i,j} v_i) \pi_k(\mathbf{b}_j^*) \right).$$

By the Gram-Schmit process, we know that $\mathbf{b}_j^* \perp \{\mathbf{b}_1, \cdots, \mathbf{b}_{j-1}\}$, and thus $\mathbf{b}_j^* \in \{\mathbf{b}_1, \cdots, \mathbf{b}_{j-1}\}^\perp$. Since π_k is defined as the orthogonal projection on the set $\{\mathbf{b}_1, \cdots, \mathbf{b}_{k-1}\}^\perp$, we then have $\pi_k(\mathbf{b}_j^*) = \mathbf{b}_j^*$ for all $k \leq j$. Indeed, since $\mathbf{b}_j^* \in \{\mathbf{b}_1, \cdots, \mathbf{b}_{j-1}\}^\perp$ and $k \leq j$, in particular, $\mathbf{b}_j^* \in \{\mathbf{b}_1, \cdots \mathbf{b}_{k-1}\}^\perp$. Thus the orthogonal projection of $\mathbf{b}_j^*$ on the set $\{\mathbf{b}_1, \cdots \mathbf{b}_{k-1}\}^\perp$ is the vector $\mathbf{b}_j^*$ itself. When $k > j$, we have $\pi_k(\mathbf{b}_j^*) = 0$ since $\mathbf{b}_j^* \perp \mathbf{b}_1, \cdots, \mathbf{b}_{j-1}$ but $\mathbf{b}_j^* \not\perp \mathbf{b}_j, \cdots, \mathbf{b}_k$.

Finally, the projection of the vector $\mathbf{c}$ can be re-written as

$$\pi_k(\mathbf{c}) = \sum_{j=k}^{n} \left((v_j + \sum_{i=j+1}^{n} \mu_{i,j} v_i) \mathbf{b}_j^* \right).$$

Question 27. *How is the enumeration tree constructed?*

The enumeration algorithm constructs a tree of depth n where each level k corresponds to the set of vectors in the projected lattice $\pi_k(\mathcal{L})$ for $k = 1, \cdots, n$ of norm less than R. The root of the tree corresponds to the zero vector since at $k = n+1$ we have $\pi_{n+1}(\mathcal{L}) = \{\mathbf{0}\}$ and the leaves are vectors of $\mathcal{L}$ of norm smaller than R since $\pi_1(\mathcal{L}) = \mathcal{L}$. Figure 2.4 illustrates an enumeration tree constructed by the algorithm.

If a vector $\mathbf{c} \in \pi_k(\mathcal{L})$ corresponds to a node, then its descendants are the vectors $\pi_{k-1}(\mathbf{c}) \in \pi_{k-1}(\mathcal{L})$ such that the norms of the projected vectors remain less than R. Note that when descending in the enumeration tree, the vectors have more coefficients as we project "less" and thus the norms increase.

At each level of the tree, the algorithm thus verifies that the projected vector has norm smaller than R, *i.e.*, that

$$||\pi_k(\mathbf{c})||^2 = \sum_{j=k}^{n} \left((v_j + \sum_{i=j+1}^{n} \mu_{i,j} v_i)^2 ||\mathbf{b}_j^*||^2 \right) \leq R^2, \tag{2.1}$$

for $k \in [1, n]$. If at some point in the enumeration process $||\pi_k(\mathbf{c})||^2 > R^2$, then the enumeration algorithm ignores the corresponding subtree since the norm can only increase as mentioned before.

Question 28. *How does the enumeration algorithm proceed?*

Concretely, the enumeration algorithm will start by considering the leaves of the tree corresponding to the vectors $\mathbf{c} = v_1 \mathbf{b}_1 \in \mathcal{L}$, updating the value $v_1 = 1, 2, \cdots$ for as long as the norm of $\mathbf{c}$ remains less than R.

When this condition is not satisfied anymore, the algorithm goes up in the enumeration tree and looks at the nodes next to $\pi_2(\mathbf{c})$ at level $k = 2$ corresponding to the vectors $v_2 \mathbf{b}_2$. Each time the coefficient v_2 is updated, thus changing the vector $\mathbf{c}$, the algorithm reconsiders the condition $||\pi_2(\mathbf{c})|| < R$. If the condition is not satisfied, the algorithm goes up to the next level.

On the other hand, if the condition is satisfied, the algorithm considers the corresponding subtree, and descends in this case back to $k = 1$ verifying if the vectors $v_1 \mathbf{b}_1 + v_2 \mathbf{b}_2$ have norms smaller than R (now updating the v_1 coefficient again). The process is repeated until an exhaustive search of all coefficients v_i for $i = 1, \cdots n$ is done. The algorithm terminates when it reaches the level $k = n+1$. By construction
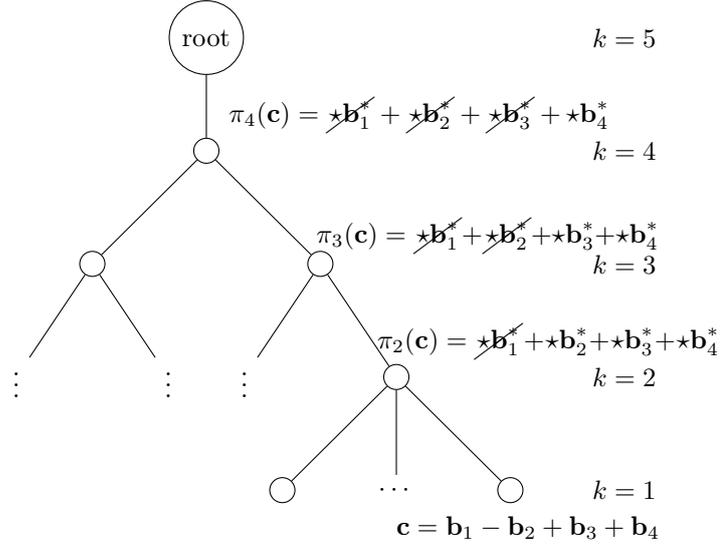
Figure 2.4: Illustration of the enumeration tree for $n = 4$. The coeffients $\star$ in front of the $\mathbf{b}_i^*$ are explicited above (in Equation 2.1 for example).

of the algorithm, whenever it considers a coefficient v_k at level k, the coefficients v_i for $i > k$ are either known or equal to 0.

The above Inequality 2.1 can be translated into the following inequality

$$\left| v_k + \sum_{i=k+1}^{n} \mu_{i,k} v_i \right| \leq \frac{\sqrt{R^2 - \sum_{j=k+1}^{n}(v_j + \sum_{i=j+1}^{n} \mu_{i,j} v_i)^2 ||\mathbf{b}_j^*||^2}}{||\mathbf{b}_k^*||}. \tag{2.2}$$

Enumerating all possible values for v_k then reduces to enumerating values in an interval $I_k = [a_k, b_k]$. Let $c_k = \sum_{i=k+1}^{n} \mu_{i,k} v_i$ be the center of the interval, then

$$a_k = \left\lceil c_k - \frac{\sqrt{R^2 - \sum_{j=k+1}^{n}(v_j + \sum_{i=j+1}^{n} \mu_{i,j} v_i)^2 ||\mathbf{b}_j^*||^2}}{||\mathbf{b}_k^*||} \right\rceil$$

and

$$b_k = \left\lfloor c_k + \frac{\sqrt{R^2 - \sum_{j=k+1}^{n}(v_j + \sum_{i=j+1}^{n} \mu_{i,j} v_i)^2 ||\mathbf{b}_j^*||^2}}{||\mathbf{b}_k^*||} \right\rfloor$$

Question 29. *How can we efficiently enumerate in the interval I_k?*

One major difference between Pohst's original algorithm (also used in Kannan's work) and Schnorr-Euchner's variant is the strategy used to enumerate within these intervals. Indeed, Pohst's strategy consists in starting at the lower bound of the interval and increasing the coefficients, *i.e.*, computing $a_k, a_k + 1, a_k + 2, \cdots, b_k$. The KFP algorithm is described in Algorithm 1 and the initialization of the interval is highlighted in blue at line 14. The exploration of the interval is given by line 18 (also in blue).

On the other hand, Schnorr-Euchner's algorithm uses a zig-zag strategy and starts at the center c_k of the interval. The values explored are then either $c_k, c_k - 1, c_k + 1, c_k - 2, c_k + 2, \cdots$ or $c_k, c_k + 1, c_k - 1, c_k + 2, c_k - 2, \cdots$ as illustrated in Figure 2.5.
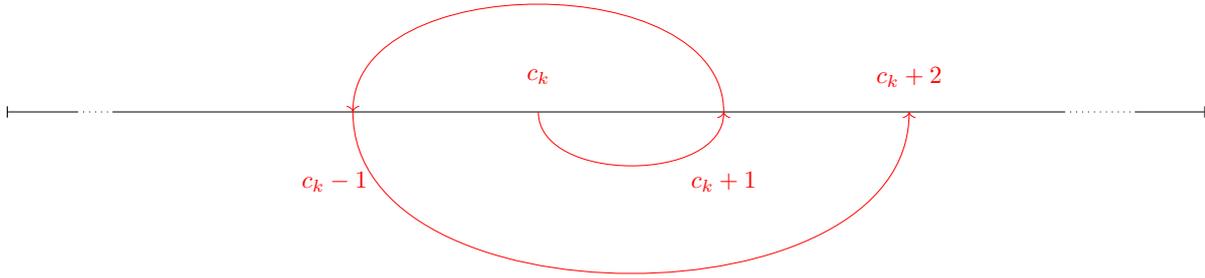
Figure 2.5: Zig-zag strategy used in Schnorr-Euchner's enumeration algorithm.

The algorithm is detailed in Algorithm 2 and this upgrade of the v_k coefficients is highlighted as $c_k + jump$ line 14 and 21. Since Schnorr-Euchner's algorithm will be used in Chapter 4 and thus explained in more details, we purposely leave a simplified pseudo-code.

Note that these enumeration procedures as written above output the list of all vectors $\mathbf{c} \in \mathcal{L}$ of norm smaller than a bound R. The leaves of the enumeration tree corresponds to these solution vectors $\mathbf{c}$. A depth-first search can be performed on the enumeration tree in order to find the shortest non-zero vector.

Question 30. *What are the main differences between these two strategies?*

Recall that the end-goal of the enumeration algorithms is to find a shortest non-zero vector of the lattice $\mathcal{L}$. Schnorr-Euchner's algorithm is an improvement on the Kannan-Fincke-Pohst algorithm as the zig-zag strategy to enumerate in the interval I_k is more efficient. By starting at the center of the interval, the values $||\pi_k(\mathbf{c})||^2$ are looked at in an incresing order. This maximizes the likelihood of quickly finding the shortest solutions such that $||\pi_k(\mathbf{c})||^2 \leq R^2$. When pruning the enumeration tree, one can also reduce the value of R once a solution is found to exclude some (useless) branches.

2.2.3 The complexity of enumeration algorithms

The complexity of the enumeration procedures is directly related to the number of loops in the algorithms. At each level k, the size of the interval I_k is roughly proportional to $2R/||\mathbf{b}_k^*||$. It corresponds to the number of nodes at level k, *i.e,* the number of internal loops. Hence, the overall number of iterations in the enumeration algorithm can be bounded by $\prod_{i=1}^{n} \lfloor \frac{2R}{||\mathbf{b}_i^*||} + 1 \rfloor$. This quantity is highly affected by the size of the norms $||\mathbf{b}_i^*||$. These norms depend on the input basis which is often pre-processed by reduction algorithms.

Question 31. *What is the impact of the quality of the input basis?*

The running-time of the enumeration algorithms varies depending on the input basis. In order to improve the running-time of these algorithms, pre-processing on the input basis can be done. Indeed, we will see in the next section that reduction algorithms make sure the norms $||\mathbf{b}_i^*||$ do not decrease too quickly, *i.e.,* that $||\mathbf{b}_i^*||^2 \leq \epsilon ||\mathbf{b}_{i+1}^*||^2$ for a small value ϵ which depends on the reduction algorithm.

Fincke and Pohst's initial algorithm uses the LLL algorithm to reduce the input basis and achieves a $2^{O(n^2)}$ running-time for a lattice of dimension n. One major difference in Kannan's work is the initial pre-processing step. Indeed, Kannan uses a much stronger reduction on the input basis by making recursive calls to the enumeration procedure in lower dimension (see the notion of HKZ-reduced basis in [HS07]). Kannan showed in [Kan87] that using this stronger reduction makes the running-time drop from $2^{O(n^2)}$ to $2^{O(n \log n)}$. A detailed analysis of Kannan's algorithm is given in [HS07]. Kannan's algorithm is however not used in practice as the pre-processing of the input basis is often more costly than the enumeration itself for dimensions we are concerned about. In [MW15], Micciancio and Walter adapted Kannan's algorithm and in particular improved the pre-processing cost in order to keep the $2^{O(n \log n)}$ complexity for lattice enumeration while only making a linear (in n) number of recursive calls during the pre-processing step.

Algorithm 1 Kannan-Fincke-Pohst enumeration algorithm	**Algorithm 2** The Schnorr-Euchner enumeration algorithm
Input: A basis $B = \{\mathbf{b}_1, \cdots, \mathbf{b}_n\}$ of $\mathcal{L}$ of dimension n, a bound R.	**Input:** A basis $B = \{\mathbf{b}_1, \cdots, \mathbf{b}_n\}$ of $\mathcal{L}$ of dimension n, a bound R.
Output: List K of vectors $\mathbf{c} \in \mathcal{L}$ of norm smaller than R.	**Output:** List K of vectors $\mathbf{c} \in \mathcal{L}$ of norm smaller than R.

1: Pre-computation: compute all Gram-Schmit coefficients $\mu_{i,j}$ for $i < j$ and the norms of the Gram-Schmidt vectors $\|\mathbf{b}_i^*\|^2$ for all $i \leq n$.	1: Pre-computation: compute all Gram-Schmit coefficients $\mu_{i,j}$ for $i < j$ and the norms of the Gram-Schmidt vectors $\|\mathbf{b}_i^*\|^2$ for all $i \leq n$.
2: $K \leftarrow \{\}$	2: $K \leftarrow \{\}$
3: $\mathbf{v} = (0, 0, \cdots, 0)$, $\eta = (0, 0, \cdots, 0)$	3: $\mathbf{v} = (1, 0, \cdots, 0)$, $\rho = (0, 0, \cdots, 0)$
4: $k = 1$	4: $k = 1$
5: **while** $k \leq n$ **do**	5: **while** $k \leq n$ **do**
6: $\quad \eta_k = (v_k + c_k)^2 \|b_k^*\|^2$	6: $\quad \rho_k = \rho_{k+1} + (v_k - c_k)^2 \|b_k^*\|^2$
7: $\quad$ **if** $\sum_{j=k}^n \eta_j \leq R^2$ **then** $\triangleright$ and $\sum_{j=k}^n \eta_j = \|\pi_k(\mathbf{c})\|^2$	7: $\quad$ **if** $\rho_k \leq R^2$ **then** $\triangleright$ and $\rho_k = \|\pi_k(\mathbf{c})\|^2$
8: $\quad\quad$ **if** $k = 1$ **then**	8: $\quad\quad$ **if** $k = 1$ **then**
9: $\quad\quad\quad \mathbf{c} = \sum_{i=1}^n v_i \mathbf{b}_i$	9: $\quad\quad\quad \mathbf{c} = \sum_{i=1}^n v_i \mathbf{b}_i$
10: $\quad\quad\quad K \leftarrow K \cup \mathbf{c}$	10: $\quad\quad\quad K \leftarrow K \cup \mathbf{c}$
11: $\quad\quad\quad v_1 = v_1 + 1$	11: $\quad\quad\quad v_1 = v_1 + 1$
12: $\quad\quad$ **else**	12: $\quad\quad$ **else**
13: $\quad\quad\quad k = k - 1$	13: $\quad\quad\quad k = k - 1$
14: $\quad\quad\quad v_k = \left\lceil c_k - \sqrt{\frac{R^2 - \sum_{j > k} \eta_j}{\|b_k^*\|^2}} \right\rceil = a_k$	14: $\quad\quad\quad v_k = \lfloor c_k \rceil$ and update *jump*
15: $\quad$ **else**	15: $\quad$ **else**
16: $\quad\quad k = k + 1$	16: $\quad\quad k = k + 1$
17: $\quad\quad v_k = v_k + 1$	17: $\quad\quad v_k = v_k + jump$
18: **return** K	18: **return** K

2.3 Reduction algorithms for Euclidean lattices

We discussed enumeration algorithms that find exact solutions to SVP and CVP. As seen in the previous section, these algorithms have super-exponential complexity, and thus are not efficient in practice when the dimension becomes too large, which is often the case in cryptography. However, in many cases in cryptanalysis, finding the exact solution to SVP is not necessary and approximate solutions are sufficient. For example, in cryptosystems such as NTRU [HPS98], the secret need not be the shortest vector of a lattice, but a reasonably short vector of it. Darmstadt Lattice 1.05-approxSVP Challenges were recently solved in [DSvW21] up to dimension 180. In this section, we discuss polynomial time algorithms that solve γ-SVP where γ is exponential in the rank of the lattice considered.

A natural approach to solving γ-SVP for lattices of dimension n is to look at reduction algorithms. Indeed, reduction algorithms transform a lattice basis in order to progressively produce a new basis of the same lattice with relatively short vectors. Schnorr [Sch87] in 1987 gave *a hierarchy of polynomial time basis reduction algorithms* which presents algorithms from the celebrated Lenstra, Lenstra, Lovász (LLL) [LLL82] algorithm to the block Korkine-Zolotareff reduction (BKZ) [SE94]. In this section, we briefly cover the LLL algorithm and its most well-known and used in practice block-variant BKZ.

2.3.1 The LLL algorithm

The LLL algorithm can be seen as an extension of Gauss algorithm for lattices of rank greater than 2. Whereas Gauss algorithm can solve exact-SVP for lattices of rank 2, LLL solves γ-SVP for $\gamma = 2^{\Theta(k)}$, where k is the rank of the lattice. We start with the definition of a reduced basis.

Definition 19 ((δ, η)-reduced basis)**.** *A basis* $B = \{\boldsymbol{b}_1, \boldsymbol{b}_2, \ldots, \boldsymbol{b}_k\} \subset \mathbb{R}^{n \times k}$ *is* (δ, η)-*reduced if there exists a number* $\delta \in (1/4, 1]$ *and* $1/2 \leq \eta < \sqrt{\delta}$ *such that the following two conditions on the Gram-*

Schmidt orthogonalized basis $\{\boldsymbol{b}_1^*, \boldsymbol{b}_2^*, \ldots, \boldsymbol{b}_k^*\} := GSO(B)$ are satisfied:

- *For $1 \leq j < i \leq k$, we have $|\mu_{i,j}| \leq \eta$,*

- *For $1 \leq i < k$, we have $\delta \|\boldsymbol{b}_i^*\|^2 \leq \mu_{i+1,i}^2 \|\boldsymbol{b}_i^*\|^2 + \|\boldsymbol{b}_{i+1}^*\|^2$,*

where the $\mu_{i,j}$ are the Gram-Schmidt coefficients.

The first condition is called size-reduction. The value of η in the original LLL paper is set to $\eta = 1/2$. The second condition is called Lovász condition and regulates the size between two consecutive reduced vectors, meaning that there is never a huge gap between $\|\mathbf{b}_i^*\|$ and $\|\mathbf{b}_{i+1}^*\|$. The original paper sets $\delta = 3/4$.

To find such bases, one can use the LLL algorithm which we recall in Algorithm 3 when $\eta = 1/2$. The LLL algorithm produces a $(\delta, 1/2)$-reduced basis for lattices with integer-valued basis vectors. More precisely, for $k \leq n$, it takes as input a basis $B := \{\mathbf{b}_1, \mathbf{b}_2, \ldots, \mathbf{b}_k\} \subset \mathbb{Z}^{n \times k}$ and outputs a $(\delta, 1/2)$-reduced basis $B_{\mathrm{red}} := \{\tilde{\mathbf{b}}_1, \tilde{\mathbf{b}}_2, \ldots, \tilde{\mathbf{b}}_k\} \subset \mathbb{Z}^{n \times k}$ such that of $\mathcal{L}(B_{\mathrm{red}}) = \mathcal{L}(B)$.

Algorithm 3 The LLL algorithm

Input: A lattice basis $B = \{\mathbf{b}_1, \mathbf{b}_2, \cdots, \mathbf{b}_k\} \subset \mathbb{Z}^{n \times k}$
Output: A $(\delta, 1/2)$-reduced basis for $\mathcal{L}(B)$.
 1: **compute** all $\mathbf{b}_i^*$ and $\mu_{i,j}$ using GSO.
 2: **for** $i = 2$ to k **do**
 3:     **for** $j = i - 1$ to 1 **do**
 4:         $\mathbf{b}_i \leftarrow \mathbf{b}_i - c_{i,j}\mathbf{b}_j$ where $c_{i,j} = \lceil \mu_{i,j} \rfloor$
 5: **swapping**:
 6: **if** $\exists i < k$ such that $\delta \|\mathbf{b}_i^*\|^2 > \mu_{i+1,i}^2 \|\mathbf{b}_i^*\|^2 + \|\mathbf{b}_{i+1}^*\|^2$ **then**
 7:     $\mathbf{b}_i \leftrightarrow \mathbf{b}_{i+1}$
 8:     **go** to 1:
 9: **return** $\mathbf{b}_1, \ldots, \mathbf{b}_k$

Remark 5. *We have presented here the classical LLL algorithm that uses rational arithmetic for the Gram-Schmidt process. It was originally noted in [Sch86] that the algorithm can be considerably sped up by using floating-point approximations. Floating-point arithmetic is used in all competitive implementations of LLL (Magma, NTL, $\mathtt{fplll}$ for example).*

Complexity of LLL. The complexity of LLL depends on the number of iterations in the algorithm and the complexity of each iteration. The number of iterations in the algorithm corresponds to the number of times two adjacent vectors are swapped. An upper bound on the number of iterations can be given thanks to a quantity known as the potential $\mathcal{D}$ of the lattice. It is shown in [MG02, Lemma 2.9] that the number of iterations is polynomial as long as δ remains less than 1.

In order to prove that the cost of each iteration is polynomial, one can look at the number of arithmetic operations performed in each loop and the size of the quantities used. It can easily be seen that the number of arithmetic operations is polynomial. As for the size of the rationals involved in the algorithm, the precision required and their magnitude is discussed in [MG02]. The complexity of LLL is summarized in the following theorem.

Theorem 7 ([MG02], Theorem 2.10)**.** *There exists a polynomial time algorithm that on input a basis $B \in \mathbb{Z}^{n \times k}$, outputs a $(\delta, 1/2)$-reduced basis of $\mathcal{L}(B)$ with parameter $\delta = (1/4) + (3/4)^{k/(k-1)}$.*

The approximation factor of LLL in theory. The LLL algorithm outputs a first vector which is short, *i.e.*, close to $\lambda_1(\mathcal{L})$ up to some approximation factor. The latter can be expressed as follows.

Lemma 2. *Let $B = \{\boldsymbol{b}_1, \boldsymbol{b}_2, \cdots, \boldsymbol{b}_k\} \subset \mathbb{R}^{n \times k}$ be an LLL (δ, η)-reduced basis. Then the shortest vector $\boldsymbol{b}_1$ satisfies $\|\boldsymbol{b}_1\| \leq \alpha^{((k-1)/2)}\lambda_1(\mathcal{L})$ or similarly $\|\boldsymbol{b}_1\| \leq \alpha^{((k-1)/4)}(\det \mathcal{L})^{1/k}$ where $\alpha = 1/(\delta - \eta^2)$.*

Proof. The second condition of a (δ, η)-reduced basis can be rewritten as $(\delta - \mu_{i+1,i}^2)||\mathbf{b}_i^*||^2 \leq ||\mathbf{b}_{i+1}^*||^2$. For any $\delta \in (1/4, 1]$, let $\alpha = \frac{1}{\delta - \eta^2}$. Then $||\mathbf{b}_i^*||^2 \leq \alpha||\mathbf{b}_{i+1}^*||^2$ as $|\mu_{i,j}| \leq \eta$. This means we have

$$||\mathbf{b}_1||^2 \leq \alpha^{i-1}||\mathbf{b}_i^*||^2 \leq \alpha^{k-1}||\mathbf{b}_i^*||^2. \tag{2.3}$$

Because this is valid for all $i = 1, 2, \cdots, k$, we must have

$$||\mathbf{b}_1|| \leq \alpha^{(k-1)/2} \min_i ||\mathbf{b}_i^*|| \leq \alpha^{(k-1)/2} \lambda_1(\mathcal{L}),$$

using the bound in Theorem 5. This proves the first relation.

As for the bound with the determinant, we have from Inequality 2.3,

$$||\mathbf{b}_1||^k \leq \prod_{i=1}^{k} \alpha^{(i-1)/2}||\mathbf{b}_i^*|| = \alpha^{\sum_{i=1}^{k}(i-1)/2} \underbrace{\prod_{i=1}^{k} ||\mathbf{b}_i^*||}_{=\det(\mathcal{L})} = \alpha^{k(k-1)/4} \det(\mathcal{L}).$$

$\square$

When choosing the initial parameters for δ and η taken as in [LLL82], that is $\delta = 3/4$ and $\eta = 1/2$, we conveniently get $\alpha = 2$. The above lemma then shows that the first vector outputted by the LLL algorithm will be a solution to γ-SVP for $\gamma = 2^{(k-1)/2}$.

However, in practice as explained in [NS06], one usually chooses $\delta \approx 1$, the limit value for which LLL runs in polynomial time, and $\eta \approx 1/2$ in order to obtain the first vector $\mathbf{b}_1$ as short as possible (*i.e.,* the value α as small as possible). In this case, one gets $\alpha \approx 4/3$. This gives an approximation factor $\gamma = \alpha^{(k-1)/2} = \left(2/\sqrt{3}\right)^{k-1}$.

The worst-case bound on the length of the shortest vector outputted by LLL can also be seen as

$$\frac{||\mathbf{b}_1||}{(\det \mathcal{L})^{1/k}} \leq (4/3)^{(k-1)/4} \lessapprox 1.075^k. \tag{2.4}$$

This bound is tight in the worst-case.

The approximation factor of LLL in practice. Despite being significantly used for cryptographic purposes for many years now, there remains a gap between the theoretical understanding of the LLL reduction algorithm and its experimental behaviour. Many works have provided experimental observations for the behaviour of LLL. In particular, the work [GN08, NS06] have shown that LLL behaves much better in practice than expected, *i.e.,* better than its proved worst-case theoretical bounds. More precisely, the authors look at the upper bound on the norm of the shortest vector outputted by LLL. In the worst case, if $\mathbf{b}_1$ is the shortest vector in the LLL-reduced basis, then we have seen that $||\mathbf{b}_1|| \leq \alpha^{(k-1)/4}(\det L)^{1/k}$. In [NS06, Heuristic 1], the authors experimentally show that this quantity $\alpha^{(k-1)/4}$ for any random basis of almost any lattice of dimension k is closer to 1.02^k, whereas the worst-case bound for practical parameters gives 1.075^k.

Another interesting approach that has emerged with the work of Madritsch and Vallée is modelling the LLL algorithm using a tool from the study of discrete dynamical systems called sandpiles models [MV10a, DKTW19].

The following Section 2.3.2 presents some work done with George Sullivan, Nadia Heninger and Daniele Micciancio from UCSD in order to better understand this gap between theory and practice. We provide an alternative experimental method to find the value 1.02 given in [NS06, Heuristic 1] as well as open questions. We refer to this value as the root approximation factor.

2.3.2 Analyzing LLL via a directed graph

Let us consider a (fixed) random full-rank lattice $\mathcal{L}$ of dimension n with a given initial basis A. Let $G_{\mathcal{L}}(V, E)$ denote the directed graph defined as follows. The nodes V correspond to LLL-reduced bases. Moreover, there exists an edge between the nodes B_i and B_j if $B_j = LLL(\sigma B_i)$, *i.e.,* one can obtain the reduced basis B_j by applying LLL to the basis σB_i which corresponds to shuffled rows of B_i for a permutation $\sigma \in S_n$. Our work attempts at answering the following questions.

Question 32. *Can the practical behavior of a basis reduction algorithm such as LLL be explained with the statistical properties of this graph? Is the LLL heuristic root approximation factor the average length of the first vector of a node?*

We start by considering the number of nodes in the graph $G_{\mathcal{L}}(V, E)$. We fix a random lattice $\mathcal{L}$ of dimension n whose basis is either an Ajtai-type basis or a Goldstein-Mayer basis. An upper bound on the number of nodes in $G_{\mathcal{L}}(V, E)$ corresponds to an upper bound on the number of LLL-reduced bases.

Claim 1 (From [Mic11]). *Let $\mathcal{L} \subset \mathbb{R}^n$ be an n-dimensional lattice. There are at most $2^{O(n^3)}$ LLL-reduced basis.*

The proof proceeds by induction and shows that there are $2^{O((n-i+1)^2)}$ possible choices for the vector $\mathbf{b}_i$ in a reduced basis. Thus, in total there are at most $\prod_{k=1}^{n} 2^{O(k^2)} = 2^{O(n^3)}$ possible LLL-reduced bases.

Recall that a graph is fully connected if every node has an edge to every other node. On the other hand, a directed graph is strongly connected if for every pair of nodes x, y there exists a path from x to y and a path from y to x. We propose the following conjecture on the connectivity of the graph $G_{\mathcal{L}}(V, E)$.

Conjecture 1. *The graph $G_{\mathcal{L}}(V, E)$ is fully connected but not strongly connected.*

This conjecture has been verified in small dimension, up to $n = 10$. Indeed, for such small dimensions, we were able to compute the entire graph, and thus enumerate all the LLL-reduced bases of a given input lattice $\mathcal{L}$. In all of our experiments, we indeed observed that the graph was fully connected. Any reduced basis can be obtained by a sequence of permutations $(\sigma_1, \sigma_2, \cdots, \sigma_i) \in S_n^i$, for some $i \geq 1$ and reductions.

On the other hand, the graph being not strongly connected implies that there exists two bases, say A and B, such that there exists a sequence of permutations allowing to reduce A to B but there is no sequence of permutations that reduces B to A.

Using random walks

In order to explore as many nodes as possible, we consider random walks on $G_{\mathcal{L}}(V, E)$. A random walk on a graph $G = (V, E)$ starts at a given node v_0 and at each step of the walk moves to a neighbor node with some probability p. The sequence of nodes constitutes a Markov chain.

We define the following random walks on the graph $G_{\mathcal{L}}(V, E)$. For a single walk j of k steps, let $\omega_j = (\sigma_1^j, \sigma_2^j, \ldots, \sigma_k^j)$ be a sequence of k permutations, where $\sigma_i^j \in S_n$ for $i = 1, \ldots, k$. Consider the following function

$$L_\sigma \colon \mathbb{Z}^{n \times n} \to \mathbb{Z}^{n \times n}$$
$$B \mapsto L_\sigma(B) = \text{LLL}(\sigma B),$$

which takes a basis B and a permutation $\sigma \in S_n$ and outputs the LLL-reduced basis of σB, where the rows of B are shuffled by σ. Moreover, we define

$$\lambda_{exp} \colon \mathbb{Z}^{n \times n} \to \mathbb{R}$$
$$B \mapsto \lambda_{exp}(B) := \left(\frac{||\mathbf{b}_1||}{\det(B)^{1/n}} \right)^{1/n}.$$

At each step $i \leq k$ of the walk, the algorithm considers a random permutation σ_i^j and computes the quantity $\lambda_{exp}(L_{\sigma_i^j}(B_i))$. The detailed description of this walk is given by Algorithm 4. In the rest of the section, we will add the subscript (i) to the value λ_{exp} to indicate that it is the value computed at the ith step.

In theory, one can then average over all possible permutations. At step i, we can rewrite the average value of the quantity λ_{exp} as:

$$\mu_A^{(i)} = \left(\frac{1}{n!} \right)^i \sum_{(\sigma_1, \ldots, \sigma_i) \in S_n^i} \lambda_{exp} \left(L_{\sigma_i} \left(L_{\sigma_{i-1}} \left(\cdots L_{\sigma_2} \left(L_{\sigma_1}(A) \right) \right) \right) \right).$$

Algorithm 4 Random walk on the graph $G_{\mathcal{L}}(V, E)$

Input: A random lattice basis $A \subset \mathbb{Z}^{n \times n}$, a number k of steps.
Output: A list L of root approximation factors.

1: $L = \{\}$
2: Reduce initial basis A with LLL, *i.e.*, $B_0 \leftarrow LLL(A)$;
3: Compute $\lambda_{exp}(B_0) \leftarrow \left(\frac{\|\mathbf{b}_1\|}{\det(B_0)^{1/n}} \right)^{1/n}$ and add to L.
4: **for** $i = 0$ to $k - 1$ **do**
5:      Select a random permutation $\sigma \in_R S_n$.
6:      Reduce shuffled basis with LLL, *i.e.*, $B_{i+1} \leftarrow L_{\sigma}(B_i)$.
7:      Compute $\lambda_{exp}(B_{i+1})$ and add to L.
8:      $i \leftarrow i + 1$.
9: **return** L.
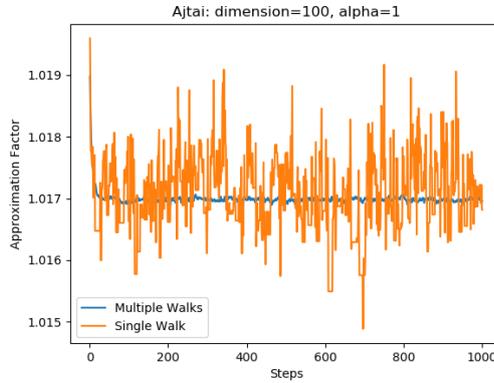
Figure 2.6: Single and multiple walks on $G_{\mathcal{L}}$ for an Ajtai random basis of dimension 100.

> **Open Question 1.** *Can one theoretically compute $\lim_{i \to \infty} \mu_A^{(i)}$ for a random basis A as a function of the dimension n and show that this limit is indeed smaller than 1.075?*

Experimental results for the approximation factor

In practice, as n grows, so does the number $n!$ of possible permutations, and thus we consider only a subset of potential walks. In our experiments, we average over T walks, for some chosen value of T. When considering T walks, one obtains T different sequences $\omega_1, \ldots, \omega_T$ of permutations, and at each step i one can compute the average value of $\lambda_{exp}^{(i)}$, *i.e.*, $\mu^{(i)} = \mathbb{E}[\lambda_{exp}^{(i)}]$. All the experiments are run using Sage's default LLL implementation.

Conjecturing the value of the approximation factor. For a fixed random lattice $\mathcal{L}$ of dimension n, we start T different random walks from the same initial random basis A. The sequences $\{\lambda_{exp}^{(i)}\}_{i=1}^{k}$, for a single walk, and $\{\mu^{(i)}\}_{i=1}^{k}$, for $T = 1000$ walks are illustrated in Figure 2.6 for an Ajtai-type random basis of dimension $n = 100$ and $k = 1000$ steps. When considering the average value of λ_{exp} over the T walks considered, one can see that the root approximation factor converges to 1.017, slightly less than 1.02.

We extensively ran such experiments for dimensions varying between 50 and 300, averaging over $T = 1000$ walks each performing $k = 250$ steps. Figure 2.7 shows the approximation factor as a function of the dimension, where the value reported is the average value obtained at the last step $k = 250$ of our walks. The curve tends to the expected approximate 1.02 value.

Based on Figure 2.7, we suggest the following conjecture where we fit the data to a function of the form $a + \frac{b}{x+c}$ for $a, b, c \in \mathbb{R}$ using Sage.

47
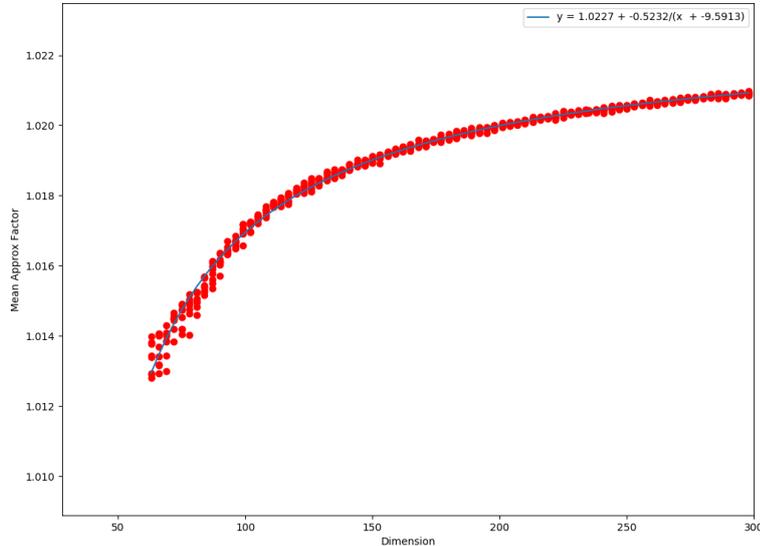
Figure 2.7: Approximation factor of LLL as a function of the dimension n of the lattice. For each dimension n, we take the value reached after 250 steps from our random walk and average over $T = 1000$ walks.

Conjecture 2. *For a fixed dimension n, we get $\lim_{i \to \infty} \mu_A^{(i)} = 1.0227 - \frac{0.5232}{n - 9.5913}$, where A is a random basis.*

2.3.3 The BKZ algorithm

The best approximation algorithm known in practice for large dimensions is the Blockwise-Korkine-Zolotarev (BKZ) algorithm, published by Schnorr and Euchner in 1994. It uses Schnorr-Euchner's enumeration algorithm as a subroutine for smaller rank sublattices. BKZ and BKZ2.0 (which uses enumeration with extreme pruning) is implemented in the `fplll` library and is extensively used to benchmark the security of lattice-based schemes.

The Schnorr-Euchner's BKZ algorithm can be seen as a generalization of LLL where instead of considering pairs of vectors, one looks at blocks of projected vectors. BKZ thus has an additional parameter $\beta \geq 2$ which corresponds to the blocksize considered. Let us consider a full-rank lattice of dimension n. For each block $B_{[i,\min(i+\beta-1,n)]}$ the algorithm ensures that the first vector is the shortest vector in that particular projected lattice by running any (approximate) SVP oracle such as an enumeration algorithm on the projected lattice. We recall the BKZ algorithm in Algorithm 5. Note that $\beta = 2$ corresponds to the LLL algorithm, where only a swapping step is considered. A BKZ-reduced basis thus satisfies a stronger condition than an LLL-reduced basis. Indeed, we have the following definition.

Definition 20 (BKZ reduced basis). *A basis $B = \{\boldsymbol{b}_1, \boldsymbol{b}_2, \cdots, \boldsymbol{b}_n\} \subset \mathbb{R}^{n \times n}$ is BKZ-reduced with blocksize $\beta \geq 2$ if it $(\delta, 1/2)$-reduced and additionally, for each $1 \leq i \leq n$, we have $||\boldsymbol{b}_i^*|| = \lambda(\mathcal{L}(B_{[i,\min(i+\beta-1,n)]}))$.*

Because BKZ calls an enumeration algorithm as a subroutine, there is an obvious tradeoff between the quality of the output basis and the running time of the algorithm. The larger the blocksize β, the more reduced the output basis is, but the cost increases as the cost of enumeration is in $2^{O(\beta^2)}$. Kannan's tighter bound mentioned previously is actually achieved in practice, see [ABF+20]. When the blocksize

Algorithm 5 The BKZ algorithm
___
Input: A lattice basis $B = \{\mathbf{b}_1, \mathbf{b}_2, \cdots, \mathbf{b}_n\} \subset \mathbb{Z}^{n \times n}$, a block-size β.
Output: A BKZ reduced basis for $\mathcal{L}(B)$.

1: Compute the Gram-Schmidt coefficients $\mu_{i,j}$ and $||\mathbf{b}_i^*||^2$ for $1 \leq j \leq i \leq n$.
2: $z \leftarrow 0$, $i \leftarrow 0$
3: LLL($\mathbf{b}_1, \cdots, \mathbf{b}_n$) $\triangleright$ LLL-reduce the basis
4: **while** $z < n - 1$ **do**
5:     $i \leftarrow (i \pmod{n-1}) + 1$ $\triangleright$ Defining the blocks
6:     $k \leftarrow \min(i + \beta - 1, n)$
7:     $h \leftarrow \min(k + 1, n)$
8:     $\mathbf{v} \leftarrow$ Oracle-SVP($B_{[i,k]}, \lambda(\mathcal{L}_{[i,k]})$) $\triangleright$ Find $\mathbf{v} \in \mathbb{Z}^{k-i+1} \setminus \mathbf{0}$ s.t. $||\pi_i(\sum_{j=i}^{k} v_j \mathbf{b}_j)|| = \lambda(\mathcal{L}_{[i,k]})$
9:     **if** $\mathbf{v} \neq (1, 0, \cdots, 0)$ **then**
10:         $z \leftarrow 0$
11:         LLL($\mathbf{b}_1, \cdots, \mathbf{b}_{i-1}, \sum_{j=i}^{k} v_j \mathbf{b}_j, \mathbf{b}_i, \cdots, \mathbf{b}_h$) $\triangleright$ Adding new vector at the beginning of the block.
12:     **else**
13:         $z \leftarrow z + 1$ $\triangleright$ LLL-reduce the next block prior to enumeration.
14:         LLL($\mathbf{b}_1, \cdots, \mathbf{b}_h$)
15: **return** ($\mathbf{b}_1, \cdots, \mathbf{b}_n$).
___

is large, *i.e.,* $\beta \geq 30$, the overall cost of BKZ is dominated by the cost of the enumeration subroutine. The output quality of BKZ and the trade-offs between the approximation factor and the running-time of BKZ are studied in [GN08, HPS11b]. The smallest approximation factor γ for which BKZ runs in polynomial time is $\gamma = 2^{O(n \log \log n / \log n)}$. A recent analysis of the BKZ algorithm is given in [LN20].

Part II

The discrete logarithm problem in finite fields

Chapter 3

Asymptotic analysis of DLP algorithms at the first boundary

In this chapter, we study the discrete logarithm problem at the boundary case between small and medium characteristic finite fields, which is precisely the area where finite fields used in pairing-based cryptosystems live. In order to evaluate the security of pairing-based protocols, we thoroughly analyze the asymptotic complexity of all the algorithms that coexist at this boundary case: the Quasi-Polynomial algorithms, the Number Field Sieve and its many variants, and the Function Field Sieve. We adapt the latter to the particular case where the extension degree is composite, and show how to lower the complexity by working in a shifted function field. All this study finally allows us to give precise values for the characteristic asymptotically achieving the highest security level for pairings. Surprisingly enough, there exist special characteristics that are as secure as general ones.

This chapter is joint work with Pierrick Gaudry and Cécile Pierrot and was published in the proceedings of the Crypto 2020 conference [DGP20].

Contents

3.1 Introduction

3.1.1 Motivation: pairing-based protocols

Pairing-based cryptography illustrates the need to consider both the discrete logarithm problems on finite fields and on elliptic curves. Let us recall that a cryptographic pairing is a bilinear and non-degenerate map $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ where $\mathbb{G}_1$ is a subgroup of $\mathcal{E}(\mathbb{F}_p)$, the group of points of an elliptic curve $\mathcal{E}$ defined over the prime field $\mathbb{F}_p$, $\mathbb{G}_2$ is another subgroup of $\mathcal{E}(\mathbb{F}_{p^n})$ where we consider an extension field and $\mathbb{G}_T$ is a multiplicative subgroup of that same finite field $\mathbb{F}_{p^n}$. To construct a secure protocol based on a pairing, one must assume that the DLPs in the groups $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ are hard.

Evaluating the security in $\mathbb{G}_1$ and $\mathbb{G}_2$ is straightforward, since very few attacks are known for DLP on elliptic curves. The most efficient known algorithm to solve the DLP in the elliptic curve setup, without any particular considerations on the curve, is Pollard's rho algorithm presented in Chapter 1.

On the contrary, the hardness of the DLP over finite fields is much more complicated to determine. Indeed, there exist many competitive algorithms that solve DLP over finite fields. When p is relatively small, Quasi-Polynomial time algorithms can be designed, but when p grows, the most efficient algorithms have sub-exponential complexity.

To construct a secure protocol based on a pairing, one must first consider a group $\mathbb{G}_T$ in which Quasi-Polynomial time algorithms are not applicable. This implies, to the best of our knowledge, that the algorithms used to solve DLP on the finite field side have an $L_{p^n}(1/3)$ complexity. Moreover, we want the complexities of the algorithms that solve DLP on both sides to be comparable. Indeed, if the latter were completely unbalanced, an attacker could solve DLP on the easier side. A natural idea is then to equalize the complexity of DLP on both sides. This requires having $\sqrt{p} = L_{p^n}(1/3)$. Hence, the characteristic p is chosen of the form $p = L_{p^n}(1/3, c_p)$ for some constant $c_p > 0$.

Yet, when the characteristic p is of this form, many algorithms coexist rendering the estimation of the hardness of DLP all the more difficult. A recent approach, followed in [Gui20] is to derive concrete parameters for a given security level, based on what the Number Field Sieve algorithm (NFS) would cost on these instances. Our approach complements this: we analyze the security of pairings in the asymptotic setup, thus giving insight for what would become the best compromise for higher and higher security levels.

The focus of this chapter is to study the complexity of all the algorithms that solve DLP at the first boundary case $p = L_{p^n}(1/3, c_p)$ and to answer the following question:

> **Question 33.** *Asymptotically what finite field $\mathbb{F}_{p^n}$ should be considered in order to achieve the highest level of security when constructing a pairing?*

This requires a significant amount of work in order to evaluate which algorithm is applicable and which one performs best. Figure 3.1 gives the general picture, without any of the particular cases that can be encountered. Figure 3.1 is actually of zoom of Figure 1.5 from Chapter 1 which focuses on our area of interest.

3.1.2 Contributions

More specifically, we propose the following contributions.

Thorough analysis of the complexity of FFS, NFS and its variants. We first give a precise methodology for the computation of the complexity of NFS and its variants at the boundary case, which
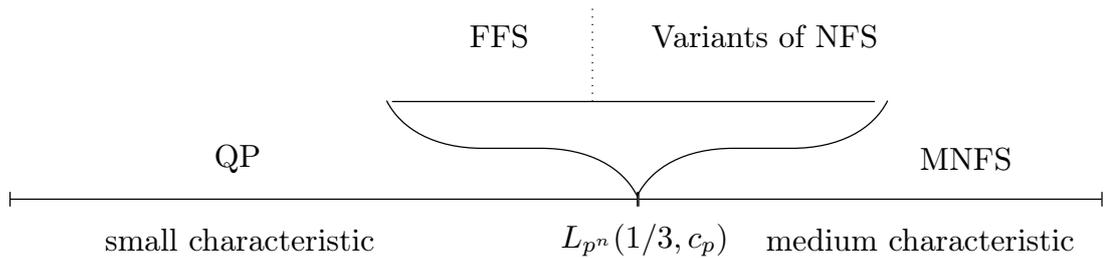
Figure 3.1: Best algorithms for DLP in small, medium characteristics and at the boundary case $p = L_{p^n}(1/3, c_p)$.

differs from the computations done in medium and large characteristics. We revisit some commonly accepted hypotheses and show that they should be considered with care. In addition, our analysis allowed us to notice some surprising facts. First of all, not all the variants of NFS maintain their $L_{p^n}(1/3)$ complexity at the boundary case. The variant STNFS, for example, has a much higher complexity in this area, and thus should not be used for a potential attack on pairings. For some special characteristics, SNFS is also not faster than MNFS, as one could expect. We also distinguish and correct errors in past papers, both in previous methodologies or computations.

FFS still remains a competitor for small values of c_p. This chapter then takes a closer look at its complexity, also fixing a mistake in the literature. Furthermore, in the case where the extension degree n is composite, we show how to lower the complexity of FFS by working in a *shifted* function field.

Crossover points between all the algorithms. This complete analysis allows us to identify the best algorithm at the boundary case as a function of c_p and give precise crossover points for these complexities. When c_p is small enough, the FFS algorithm remains the most efficient algorithm outperforming NFS and all of its variants. When the extension degree n is prime, and the characteristic has no special form, the algorithm MNFS outperforms FFS when $c_p \geq 1.23$. When n is composite or p taken of a special form, variants such as exTNFS and SNFS give crossover points with lower values for c_p, given in this chapter.

Moreover, we compare the complexity of FFS and the complexity of the Quasi-Polynomial algorithms. Since the crossover point occurs when p grows slightly slower than $L_{p^n}(1/3)$, we introduce a new definition in order to determine the exact crossover point between the two algorithms.

Security of pairings. All the work mentioned above allows us to answer the following question: asymptotically what finite field $\mathbb{F}_{p^n}$ should be considered in order to achieve the highest level of security when constructing a pairing? To do so, we justify why equating the costs of the algorithms on both the elliptic curve side and the finite field side is correct and argue that in order for this assumption to make sense, the complete analysis given in this chapter was necessary. Finally, we give the optimal values of c_p for the various forms of p and extension degree n, also taking into account the so-called ρ-value of the pairing construction. Surprising fact, we were also able to distinguish some special characteristics that are asymptotically as secure as characteristics of the same size but without any special form.

Asymptotic complexities versus practical estimates. The fact that STNFS is asymptotically no longer the best algorithm for optimally chosen pairing-friendly curves is not what could be expected from the study of [Gui20], where fixed security levels up to 192 bits are considered. This could be interpreted as a hint that cryptanalysts have not yet reached some steady state when working at a 192-bit security level. To sum it up, evaluating the right parameters for relevant cryptographic sizes (e.g. pairings at 256-bit of security level) is still hard: estimates for lower sizes and asymptotic analysis do not match, and there is no large scale experiment using TNFS or variants to provide more insight.

3.2 The FFS algorithm at the boundary case

We consider a finite field of the form $\mathbb{F}_{p^n}$, where p is the characteristic and n the extension degree. From now on, we set $Q = p^n$. Since our analysis is asymptotic, any factor that is ultimately hidden in the $o(1)$ notation of the L_Q expression is ignored. Furthermore, inequalities between quantities should be understood asymptotically, and up to negligible factors.

Recall that the characteristic $p = L_Q(1/3, c_p)$ and from the definition of the L_Q-notation we deduce $n = \frac{1}{c_p} \left(\frac{\log Q}{\log \log Q} \right)^{2/3}$.

3.2.1 Complexity analysis of FFS

We have seen in Chapter 1 how the Function Field Sieve algorithm instantiates itself in the general framework of index calculus algorithms. In this section, we focus on a simpler variant of FFS introduced by Joux and Lercier [JL06] which does not require the theory of function fields.

Joux-Lercier setup for FFS.

The algorithm starts by choosing two univariate polynomials $f_1, f_2 \in \mathbb{F}_p[x]$ of degrees n_1, n_2 respectively such that $n_1 n_2 \geq n$ and there exists a degree-n irreducible factor I of $x - f_2(f_1(x))$. Then, let us set $y = f_1(x)$, which constitutes a first equation. In the target finite field represented as $\mathbb{F}_p[x]/(I(x))$, we therefore also have the second equation $x - f_2(y)$ or similarly $x = f_2(y) \pmod{I}$. The factor basis $\mathcal{F}$ is defined as the set of all univariate, irreducible, monic polynomials of degree D for some constant D, in x and y. As usual, the sieving phase computes multiplicative relations among elements of the factor basis, that become linear relations between discrete logarithms. We sieve over bivariate polynomials $\phi(x, y)$ of the form $\phi(x, y) = A(x)y + B(x)$, where A, B have degrees d_1, d_2 and A is monic. As an element of the finite field, this can be rewritten either as a univariate polynomial in x, namely $F_x(x) = \phi(x, f_1(x))$, or as a univariate polynomial in y, namely $F_y(y) = \phi(f_2(y), y)$. We get a relation if both $F_x(x)$ and $F_y(y)$ are D-smooth. Once enough relations are collected, the linear algebra and descent steps are performed.

Complexity analysis.

Our description of the complexity analysis of FFS is based on [SS16a]. However, we slightly deviate from their notations as theirs lead to wrong complexities (see Section 3.2.3 for details).

First, a parameter $\Delta \geq 1$ is chosen which controls the balance between the degrees of the defining polynomials f_1 and f_2. We select f_1 and f_2 of degree $\deg f_1 = n_1 = \lceil \sqrt{n\Delta} \rceil$ and $\deg f_2 = n_2 = \lceil \sqrt{n/\Delta} \rceil$. Since we use the pinpointing technique, which we recall in Section 3.2.2, we also enforce $f_1(x) = x^{n_1}$ or $f_2(y) = y^{n_2}$, depending on which side we want to pinpoint with.

For the analysis, the smoothness bound $D \geq 1$ is also fixed. Once c_p, Δ and D are fixed, we look at the complexity of the three steps of the algorithm. For the linear algebra, the cost $\mathcal{C}_{\text{linalg}}$ is quadratic in the size of the factor basis, *i.e.*, p^D, and thus we get $\mathcal{C}_{\text{linalg}} = L_Q(1/3, 2c_p D)$. For the other steps, the complexity depends on bounds d_1 and d_2 on the degree of the polynomials A and B, used to find relations. Asymptotically, no improvement is achieved by taking $d_1 \neq d_2$. Therefore, we set the following notation: $d_{12} = d_1 = d_2$. However, the value d_{12} is not necessarily the same for the sieving and descent steps.

Analysis of the sieving step. Asymptotically, we have $\deg F_x = n_1$ and $\deg F_y = d_{12} n_2$. Note that in truth $\deg F_x = n_1 + d_{12}$ but d_{12} is a constant, hence can be ignored, since n_1 goes to infinity. From these values and the smoothness bound D, we apply Flajolet, Gourdon and Panario's theorem [PGF98]. While the general theorem was given in Chapter 1, Corollary 1, we recall that the logarithm of the D-smoothness probability of a polynomial f is given by $\log P_f = \frac{\deg f}{D} \log \left(\frac{D}{\deg f} \right)$.

We then deduce the following smoothness probabilities P_{F_x} and P_{F_y} using the definitions of n_1, n_2 and the expression of n as a function of $\log Q$:

$$P_{F_x} = L_Q \left(\frac{1}{3}, \frac{-\sqrt{\Delta}}{3D\sqrt{c_p}} \right), \quad \text{and} \quad P_{F_y} = L_Q \left(\frac{1}{3}, \frac{-d_{12}}{3D\sqrt{c_p\Delta}} \right).$$

The number of (A, B)-pairs to explore before having enough relations is then $P_{F_x}^{-1} P_{F_y}^{-1}$ times the size of the factor base. To have enough relations, one needs to explore a sufficiently large amount of sieving polynomials $\phi(x, y)$ and this is feasible only if the degree d_{12} of A and B is large enough. Recalling that A is monic, this leads to the following constraint: $p^{2d_{12}+1} \geq P_{F_x}^{-1} P_{F_y}^{-1} p^D$, where $2d_{12} + 1$ is the number of undetermined coefficients in the polynomials $\phi(x, y)$.

Furthermore, using the pinpointing technique allows to find relations faster than exploring them all. We simply state here that the cost per relation with pinpointing is $\min(P_{F_x}^{-1}, P_{F_y}^{-1}) + p^{-1} P_{F_x}^{-1} P_{F_y}^{-1}$, more details being given below. The total cost $\mathcal{C}_{\text{siev}}$ for constructing the whole set of relations is then this quantity multiplied by p^D and we get

$$\mathcal{C}_{\text{siev}} = p^{D-1} P_{F_x}^{-1} P_{F_y}^{-1} + p^D \min(P_{F_x}^{-1}, P_{F_y}^{-1}). \tag{3.1}$$

Analysis of the descent step. During the descent step, it can be shown that the bottleneck happens at the leaves of the descent tree, *i.e.,* when descending polynomials of degree $D + 1$, just above the smoothness bound. The smoothness probabilities P_{F_x} and P_{F_y} take the same form as for the sieving step, but the feasibility constraint and the cost are different. Since we only keep the (A, B)-pairs for which the degree $D + 1$ polynomial to be descended divides the corresponding norm, we must subtract $D + 1$ degrees of freedom in the search space, which becomes $p^{2d_{12}-D}$. The descent step will therefore succeed under the following constraint: $p^{2d_{12}-D} \geq P_{F_x}^{-1} P_{F_y}^{-1}$. Indeed, recall from the descent analysis in Chapter 1 that the cost of descending one element is $P_{F_x}^{-1} P_{F_y}^{-1}$, as only one relation is enough. Finally, the number of nodes in a descent tree is polynomial, and the total cost $\mathcal{C}_{\text{desc}}$ of this step remains $\mathcal{C}_{\text{desc}} = P_{F_x}^{-1} P_{F_y}^{-1}$.

Overall complexity. To obtain the overall complexity for a given value of c_p, we proceed as follows: for each $\Delta \geq 1$ and $D \geq 1$, we look for the smallest value of $d_{12} \geq 1$ for which the feasibility constraint is satisfied for sieving and get the corresponding $\mathcal{C}_{\text{siev}}$; then we look for the smallest value of $d_{12} \geq 1$ such that the feasibility constraint is satisfied for the descent step and get the corresponding $\mathcal{C}_{\text{desc}}$. The maximum complexity amongst the three costs gives a complexity for these values of Δ and D. We then vary Δ and D and keep the lowest complexity. The result is shown in Figure 3.2, where the colors indicate which step is the bottleneck for each range of c_p values.
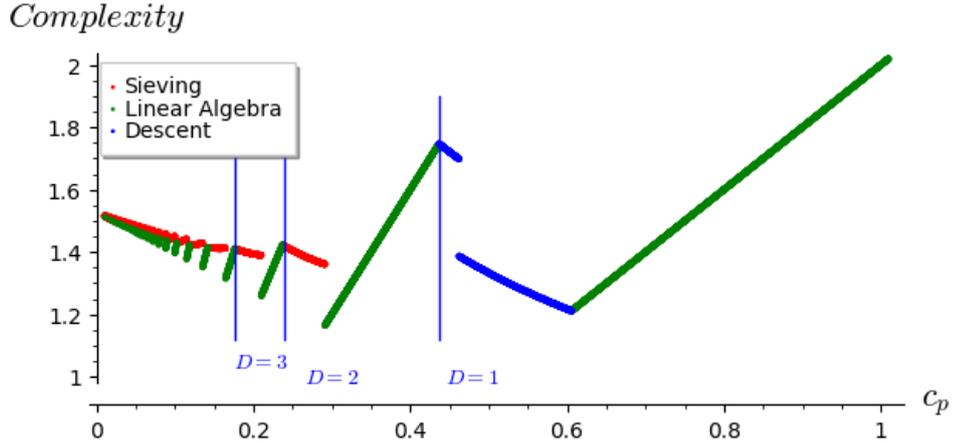
Figure 3.2: The complexity of FFS at the boundary case and the dominant phase as a function of c_p, obtained after fixing the error in [SS16a].

Remark 6. *The plot of the complexity of FFS is intriguing. Indeed, it is far from smooth and moreover presents patterns that can be explained.*

- *For small values of c_p, i.e., $c_p \lessapprox 0.4$, the dominant cost of FFS alternates between sieving and linear algebra. More surprisingly, there exist jumps between the two costs and a sudden increase in*

the overall complexity due to the cost of sieving when looking at decreasing values of c_p. We now explain this phenomenon.

When the linear algebra cost is predominant, it naturally decreases when c_p decreases. Indeed, recall that this cost is simply given by $L_{p^n}(1/3, 2Dc_p)$. Moreover, recall that for the sieving step there exists a constraint on the degrees of A and B. The latter needs to be large enough to produce enough relations. The discontinuities in the complexity and the sudden increases in the cost of sieving correspond to the values of c_p for which the degree d_{12} of A and B must be increased in order for the constraint (inequality) to be satisfied. Increasing d_{12} thus increases the cost of sieving which becomes predominant until the linear algebra takes over again due this time to the increase of the value D. The parameter D is the upper bound on the degree of the polynomials in the factor basis and if it is not increased when $c_p \to 0$, the algorithm fails to find any relations. Increasing D increases the cost of linear algebra but decreases the costs of sieving and descent (this can be seen with the formulas given above).

- *Why is the descent suddenly more costly for $0.4 \lessapprox c_p \lessapprox 0.6$? This area corresponds to c_p values for which the optimal complexity is given when $D = 1$. The cost of linear algebra is thus simply $2c_p$. As for comparing the costs of sieving and descent, we recall that the degree d_{12} is not necessarily set the same for both phases and depends on the constraints given above. For this range of c_p values, the degree d_{12} is smaller for sieving and thus results in a lower cost. Hence, the dominant cost becomes the one for the descent. Note that when $c_p > 0.6$, both costs are dominated by $2c_p$, i.e., the linear algebra.*

3.2.2 The pinpointing technique

In [Jou13], Joux introduces a trick that allows to reduce the complexity of the sieving phase. We recall the main idea in the particular case where we use pinpointing on the x-side and when $d_{12} = 1$.

In this case, the polynomial f_1 is restricted to the particular form $f_1(x) = x^{n_1}$. For a pair of polynomials $A(x) = x + a$, $B(x) = bx + c$, the polynomial $F_x(x)$ becomes $F_x(x) = \phi(x, f_1(x)) = x^{n_1+1} + ax^{n_1} + bx + c$. One can then perform the change of variable $x \mapsto tx$ for t in $\mathbb{F}_p^*$, and, making the expression monic, one gets the following polynomial

$$G_t(x) = x^{n_1+1} + at^{-1}x^{n_1} + bt^{-n_1}x + ct^{-n_1-1}.$$

If $F_x(x)$ is D-smooth, so is $G_t(x)$, which corresponds to $F_x(x)$ with the (A, B)-pair given by $A(x) = x + at^{-1}$ and $B(x) = (bt^{-n_1}x + ct^{-n_1-1})$.

To evaluate $\mathcal{C}_{\text{siev}}$ using the pinpointing technique, we first need to consider the cost of finding the initial polynomial, *i.e.*, an (A, B)-pair such that $F_x(x)$ is D-smooth. Then, varying $t \in \mathbb{F}_p^*$ allows to produce $p-1$ pairs which, by construction, are also smooth on the x-side. We then need to check for each of them if $F_y(y)$ is also smooth. The total cost is thus $P_{F_x}^{-1} + p$, and the number of relations obtained is pP_{F_y}. Finally the cost per relation is $P_{F_y}^{-1} + (pP_{F_x}P_{F_y})^{-1}$.

By symmetry, the only difference when doing pinpointing on the y-side is the first term which is replaced by $P_{F_x}^{-1}$. Choosing the side that leads to the lowest complexity, and taking into account that we have to produce p^D relations leads to the overall complexity for the sieving step given in Equation (3.1) which we recall here

$$\mathcal{C}_{\text{siev}} = p^{D-1}P_{F_x}^{-1}P_{F_y}^{-1} + p^D \min(P_{F_x}^{-1}, P_{F_y}^{-1}).$$

3.2.3 Fixing a rounding bug in the FFS analysis of [SS16a]

Section VIII of [SS16a] contains an asymptotic analysis of FFS in the boundary case. Their analysis contains a problem leading to slightly wrong complexities. This has now been corrected by the authors in the last eprint version of their article, as we discussed this issue with them. The source of the mistake is the fact that d_1 and d_2, the bound of the degrees of the polynomials $A(x)$ and $B(x)$ that are used to define the sieving polynomial $\phi(x, y) = A(x)y + B(x)$, must be integers. Since in a whole range of values for c_p these degrees can be as small as a few units (and sometimes their optimal value is actually 1), forgetting to round the values to integers can lead to significant differences.

More precisely, in the proof of Proposition 3 of page 2248, the optimal value of e^* is rounded to an integer. But then, Δ^* which is the new name of d_1 and d_2 in this portion of the paper is taken to be $\Delta^* = (e^* - 1)/2$ which is an integer only if e^* is odd, and there is no reason for this to be true. The same problem occurs on page 2249, where a study similar to the one of Proposition 3 is performed.

One possibility would be to take $d_2 = d_1 + 1$ when the parity of e^* is not the good one. But then the degree of the $F_y(y)$ polynomials is no longer $n_2 d_1 + 1$, but $\max(n_2 d_1 + 1, n_2 d_2)$, so that asymptotically, this degree can be approximated by $\max(d_1, d_2) n_2$. Therefore, unbalancing the degrees of d_1 and d_2 does not solve the issues.

To give a better illustration of the problem, we follow the analysis of [SS16a] for the value $c_p = (1/6)^{1/3}$ which, according to this paper, yields the best overall complexity.

For this value of c_p, the formulae leading to the lowest cost are the ones of Section VIII.B on page 2249, with the parameter $D = 1$. The equations (37), (38), (39) lead to the following values:

$$e^*_{(1,\mathrm{des})} = 2; \ \Delta^*_{(1,\mathrm{des})} = 3/2; \ e^*_{(1,\mathrm{rel})} = 3; \ \Delta^*_{(1,\mathrm{rel})} = 1;$$

$$\delta_1 = \max(0.9172\ldots, 1.1006\ldots, 1.1006\ldots) = 1.1006\ldots$$

We see here clearly that the value of $\Delta^*_{(1,\mathrm{des})}$ is not an integer, which contradicts the convention taken on page 2247: *Then Δ_{rel} (resp. Δ_{des}) gives the degrees of $A(x)$ and $B(x)$ for relation collection (resp. descent).*

Therefore, for $c_p = (1/6)^{1/3}$ we have to take $\Delta^*_{(1,\mathrm{des})} = 2$ instead of $3/2$, which increases the overall cost of the descent to 1.27 instead of 1.14.

With our new analysis as presented in Section 3.2, the value of c_p for which we obtain the best overall complexity is now $c_p = (2/81)^{1/3}$, where the cost is $L_Q(1/3, 4\,(2/81)^{1/3}) \approx L_Q(1/3, 1.16)$.

3.2.4 Improving the complexity of FFS in the composite case

We are able to lower the complexity of FFS when the extension degree n is composite. This case often happens in pairings for efficiency reasons.

Let $n = \eta\kappa$. This means we can rewrite our target field as $\mathbb{F}_{p^n} = \mathbb{F}_{p^{\eta\kappa}} = \mathbb{F}_{p'^\eta}$, where $p' = p^\kappa$. Note that this would not work in the NFS context because p' is no longer a prime. From $p = L_Q(1/3, c_p)$, we obtain $p' = L_Q(1/3, \kappa c_p)$. Thus looking at the complexity of FFS in $\mathbb{F}_{p^n}$ for some $c_p = \alpha$ is equivalent to looking at the complexity of FFS in $\mathbb{F}_{p'^\eta}$ at some value $c_{p'} = \kappa\alpha$. This corresponds to a shift of the complexity by a factor of κ.

More generally, assume n can be decomposed into a product of multiple factors. For each factor κ of n, one can consider the target field $\mathbb{F}_{p'^r}$, where $p' = p^\kappa$ and $r = n/\kappa$. This gives rise to a new complexity curve $\mathcal{C}_\kappa$, shifted from the original one by a factor of κ. One can then consider the final curve $\mathcal{C} = \min_{\kappa \geq 1} \mathcal{C}_\kappa$, that assumes that n has many small factors. This lowers the complexity of FFS for small values of c_p as can be seen in Figure 3.3.

> **Question 34.** *Morally, what does this imply?*

When the degree D becomes too small to find all the relations required, instead of increasing it, one can consider working with a small extension of $\mathbb{F}_p$ as a base field for FFS.

One of the most significant examples is when $c_p = (1/6) \times (2/81)^{1/3} = 0.049$. The FFS complexity is $L_Q(1/3, 1.486)$ in this case, while if n is a multiple of 6, we can use $p' = p^6$, so that we end up at the point where FFS has the lowest complexity, and we reach $L_Q(1/3, 1.165)$.

More generally, even when the characteristic is small, if $n = \eta\kappa$ is composite we can work with $\mathbb{F}_{p^\kappa}$ as a base field, and if p^κ has the appropriate size we can have a complexity that is lower than the $L_Q(1/3, (32/9)^{1/3})$ of the plain FFS in small characteristic. The optimal case is given for the value $\kappa = (2/81)^{1/3} n^{1/3} (\log_p n)^{2/3}$. This strategy is very similar to the extended Tower NFS technique where we try to emulate the situation where the complexity of NFS is the best (see Chapter 4).

Now that we have given an analysis of the Function Field Sieve at the boundary case, we turn our attention to the Number Field Sieve and its variants.
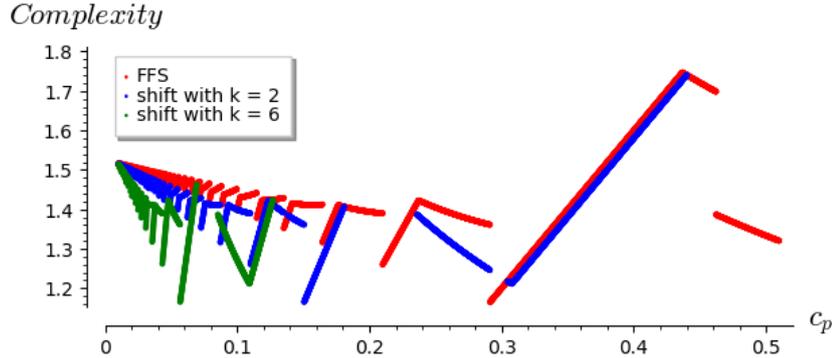
Figure 3.3: Assuming n has appropriate factors, the lowered complexity of FFS for small values of c_p when considering shifts. In this plot, we consider $\kappa = 2, 6$. We only plot points of the curves $\mathcal{C}_2$ and $\mathcal{C}_6$ which are lower than the original FFS curve.

3.3 Tools for the analysis of NFS and its variants

The main difficulty when evaluating the complexity of NFS is the amount of parameters that influence in a non-trivial way the running time or even the termination of the algorithm. In this section we explain our methodology to find the set of parameters leading to the fastest running time. We do not consider the space complexity. Indeed, in all the variants of NFS under study, the memory requirement is dominated by the space required to store the matrix of relations, which is equal (up to logarithmic factors) to the square root of the running time to find a kernel vector in this matrix.

3.3.1 General methodology

Parameters and their constraints

As often done in an asymptotic complexity analysis, even if parameters are assumed to be integers, they are considered as real numbers. This is a meaningful modelling as long as the numbers tend to infinity since the rounding to the nearest integer will have a negligible effect. In some of the variants, however, some integer parameters remain bounded. This is the case for instance of the (r, k) parameters in MNFS-$\mathcal{A}$, detailed in Section 3.5. We call continuous parameters the former, and discrete parameters the latter.

The analysis will be repeated independently for all values of the discrete parameters, so that we now concentrate on how to optimize the continuous parameters for a given choice of discrete parameters. We call a set of parameters valid if the algorithm can be run and will finish with a high probability. Many parameters are naturally constrained within a range of possible values in $\mathbb{R}$. For instance, a smoothness bound must be positive. In addition, one must consider another general constraint to ensure the termination of the algorithm: the number of relations produced by the algorithm for a given choice of parameters must be larger than the size of the factor basis. We will refer to this constraint as the Main Constraint in the rest of the chapter.

This inequality can be turned into an equality with the following argument (similarly as in Section 3.2, equality is up to asymptotically negligible factors). Assume a set of parameters gives a minimum running time, and for these parameters the number of relations is strictly larger than the size of the factor basis. Then, by reducing the bound on the coefficients of the polynomials ϕ used for sieving, one can reduce the cost of the sieving phase, while the costs of linear algebra and individual logarithm steps stay the same. Therefore, one can construct a new set of parameters with a smaller running time.

The costs of the three phases

Let $\mathcal{C}_{\mathrm{siev}}, \mathcal{C}_{\mathrm{linalg}}$ and $\mathcal{C}_{\mathrm{desc}}$ be the costs of the three main phases of NFS. The overall cost of computing a discrete logarithm is then the sum of these three quantities. Up to a constant factor in the running time,

the optimal cost can be deduced by minimizing the maximum of these three costs instead of their sum. Given the form of the formulas in terms of the parameters, this will be much easier to handle.

A natural question that arises is whether, at the optimum point, one cost "obviously" dominates the others or on the contrary is negligible. The two following statements were previously given without justification and we correct this issue here. First, the best running time is obtained for parameters where the linear algebra and the sieving steps take a similar time. We explain why there is no reason to believe this assumption is necessarily true. Secondly, the cost of the individual logarithm step is negligible. We justify this in this setting with a theoretical reason.

Equating the cost of sieving and linear algebra. In the most simple variant of NFS for solving the discrete logarithm in a prime field using two number fields, the best complexity is indeed obtained at a point where linear algebra and sieving have the same cost. However, we would like to emphasize that this is not the result of an "obvious" argument. Let us assume that the linear algebra is performed with an algorithm with complexity $O(N^\omega)$, where ω is a constant. The matrix being sparse, the only lower bound we have on ω is 1, while the best known methods [Cop94] give $\omega = 2$. By re-analyzing the complexity of NFS for various values of ω, we observe that the optimal cost is obtained at a point where the linear algebra and the sieving have similar costs only when $\omega \geq 2$. Were there to be a faster algorithm for sparse linear algebra with a value of ω strictly less than 2, the complexity obtained with $\mathcal{C}_{\mathrm{siev}} = \mathcal{C}_{\mathrm{linalg}}$ would not be optimal. Therefore, any "obvious" argument for equating those costs should take into account that the current best exponent for sparse linear algebra is 2.

Negligible cost of individual logarithm step. We will argue that in NFS and any of its variants, the cost of individual logarithms is always negligible compared to the cost of sieving, thus simplifying the analysis.

As explained previously, the individual logarithm phase consists of two steps: a smoothing step and a descent by special-q. We refer to [FGHT17, Appendix A] where a summary of several variants is given, together with the corresponding complexities. The smoothing part is somewhat independent and has a complexity in $L_Q(1/3, 1.23)$, that is lower than all the other complexities. Note however that [FGHT17] does not cover the case where a discrete logarithm in an extension field is sought. The adaptation can be found in [BP14, Appendix A] and the complexity remains the same. For the special-q descent step, the analysis of [FGHT17] does not need to be adapted, since all the computations take place at the level of number fields. Using sieving with large degree polynomials, it is shown that all the operations except for the ones at the leaves of the descent trees take a negligible time $L_Q(1/3, o(1))$. Finally the operations executed at the leaves of the tree are very similar to the ones performed during sieving to find a single relation. Therefore they also take a negligible time compared to the entire sieving step that must collect an $L_Q(1/3)$ sub-exponential quantity of relations, while we only require a polynomial quantity for the descent. As a consequence, in the context of NFS and its variants, the individual logarithm phase takes a much smaller time than the sieving phase.

Overall strategy for optimizing the complexity

First we fix values for the discrete parameters. Since these values are bounded in our model, there are only finitely many choices. We then apply the following recursive strategy where all the local minima of the parameters encountered are compared in the end and the smallest is returned. The strategy executes the following two steps. First, in the subvariety of valid parameters satisfying the Main Constraint, search for local minima of the cost assuming $\mathcal{C}_{\mathrm{siev}} = \mathcal{C}_{\mathrm{linalg}}$. Then, recurse on each plausible boundary of the subvariety of parameters.

In order for our analysis to remain as general as possible, we have also considered the case where the costs of sieving and linear algebra are not equal. We then look for local minima for $\mathcal{C}_{\mathrm{siev}}$ and see if this results in a lower complexity. We do not detail this case since this situation has not occurred in our analyses, but we insist on the necessity to perform these checks.

We emphasize that we have to first look for minima in the interior of the space of valid parameters and then recurse on its boundaries. This is imposed by the technique we use to find local minima. Indeed, we assume that all the quantities considered are regular enough to use Lagrange multipliers. However, this technique cannot be used to find a minimum that would lie on a boundary. This is the case for example of STNFS as explained in Section 3.5.3.

In general, only few cases are to be considered. For instance, except for a few polynomial selection methods, there are no discrete parameters. Also, boundary cases are often non-plausible, for example, when the factor base bound tends to zero or infinity. Some cases are also equivalent to other variants of NFS, for instance when the number of number fields in MNFS goes to zero, the boundary case is the plain NFS.

Notations. In the analysis of NFS and its variants, parameters that grow to infinity with the size Q of the finite field must be chosen with the appropriate form to guarantee an overall $L_Q(1/3)$ complexity. We summarize in Table 3.1 the notations used for these parameters, along with their asymptotic expressions. For convenience, in order to have all the notations at the same place, we include parameters that will be introduced later in this chapter.

Notation	Asympt. expression	Definition
General parameters		
p	$L_Q(1/3, c_p)$	Characteristic of finite field $\mathbb{F}_Q$, where $Q = p^n$
n	$\frac{1}{c_p}\left(\frac{\log Q}{\log\log Q}\right)^{2/3}$	Exponent of finite field $\mathbb{F}_Q$, where $Q = p^n$
B	$L_Q(1/3, c_B)$	Smoothness bound
t	$\frac{c_t}{c_p}\left(\frac{\log Q}{\log\log Q}\right)^{1/3}$	Degree of the sieving polynomials ϕ
A	$(\log Q)^{c_A c_p}$	Bound on coefficients of ϕ. Note $A^t = L_Q(1/3, c_A c_t)$
P	$L_Q(1/3, p_r)$	Probability that ϕ leads to a relation
MNFS parameters		
V	$L_Q(1/3, c_V)$	Number of number fields in MNFS
B'	$L_Q(1/3, c_{B'})$	Second smoothness bound for asymmetric MNFS
P_1	$L_Q(1/3, p_{r1})$	Probability of smoothness in the first number field
P_2	$L_Q(1/3, p_{r2})$	Probability of smoothness in any other number field
Other parameters		
d	$\delta\left(\frac{\log Q}{\log\log Q}\right)^{2/3}$	Degree of polynomial in the case of JLSV2
η	$c_\eta\left(\frac{\log Q}{\log\log Q}\right)^{1/3}$	Factor of n in the case of (ex)TNFS; $\deg h = \eta$
κ	$c_\kappa\left(\frac{\log Q}{\log\log Q}\right)^{1/3}$	Other factor of n in the case of (ex)TNFS; $c_\kappa = \frac{1}{c_p c_\eta}$

Table 3.1: Notations and expressions for most of the quantities involved in the analysis of NFS and its variants.

3.3.2 Smoothness probability

During the sieving phase, we search for B-smooth norms, *i.e.*, norms of ideals that would decompose into prime ideals with norm smaller than B. The set of prime ideals with norm smaller than B is known as the factor basis. A key assumption in the analysis is that the probability of a norm being smooth is the same as that of a random integer of the same size. This allows us to apply the theorem by Canfield-Erdős-Pomerance [**?**]. While the general theorem was given in Chapter 1, we use the following specific version stated using the L_Q-notation:

Corollary 3. *Let $(\alpha_1, \alpha_2, c_1, c_2)$ be four real numbers such that $1 > \alpha_1 > \alpha_2 > 0$ and $c_1, c_2 > 0$. Then the probability that a random positive integer below $L_Q(\alpha_1, c_1)$ splits into primes less than $L_Q(\alpha_2, c_2)$ is given by*

$$L_Q\left(\alpha_1 - \alpha_2, (\alpha_1 - \alpha_2)c_1 c_2^{-1}\right)^{-1}.$$

The norms are estimated based on their expressions as resultants. In the classical (non-tower) version of NFS, for a given candidate ϕ, the norm N_i in the i-th number field given by f_i takes the form $N_i(\phi) = c\,\mathrm{Res}(f_i, \phi)$, where c is a constant coming from the leading coefficient of f_i that can be considered smooth (possibly by including its large prime factors in the factor basis).

The usual definition of the resultant is given as the determinant of the Sylvester matrix, *i.e.*,

$$Res(f_i, \phi) = \sum_{\sigma \in S_{\deg f_i + \deg \phi}} \epsilon(\sigma) \prod_{i=1}^{\deg f_i + \deg \phi} a_{i\sigma(i)},$$

with $\epsilon(\sigma)$ the signature of the permutation and $a_{i,j}$ the coefficients of the Sylvester matrix. Since $|S_{\deg f_i + \deg \phi}| = (\deg f_i + \deg \phi)!$, a naive bound on the resultant is

$$|\operatorname{Res}(f_i, \phi)| \leq (\deg f_i + \deg \phi)! ||f_i||_\infty^{\deg \phi} ||\phi||_\infty^{\deg f_i}$$

In our context, we use another bound that follows from Hadamard's inequality (see [BL10, Theorem 7]):

$$|\operatorname{Res}(f_i, \phi)| \leq (d+1)^{(t-1)/2} t^{d/2} ||f_i||_\infty^{(t-1)} ||\phi||_\infty^d,$$

where $d = \deg f_i$ and $t - 1 = \deg \phi$. Note that in our setting, d must be larger than n which is roughly in $(\log Q)^{2/3}$, while t is in $(\log Q)^{1/3}$, so that the factor $(d+1)^{(t-1)/2}$ will be negligible but the factor $t^{d/2}$ will not.

A note on Kalkbrener's corollary. Recent papers including [JP14b, BGGM15, SS16c, SS16b] have mentioned a result from Kalkbrener [Kal97] to upper bound the value of the combinatoric term $(\deg f_i + \deg \phi)!$. Recall that this term corresponds to the number of permutations which gives a non-zero product in the definition of the resultant. In [Kal97], Theorem 2 counts the number of monomials in the Sylvester matrix. However, two permutations can give the same monomial, and thus the number of permutations is not bounded by the number of monomials. We emphasize that this result cannot be used this way; this error leads to wrong (and underestimated) complexities. Indeed combinatorial terms cannot be neglected at the boundary case.

When analyzing tower variants (see [KB16, Lemma 1] and [SS19, Equation 5]), the ring R is $\mathbb{Z}[\iota]/h(\iota)$, and in all cases, the optimal value for the degree of $\phi(X)$ is 1 (i.e. $t = 2$, in the general setting). A polynomial ϕ is therefore of the form $\phi(X) = a(\iota) + b(\iota)X$, where a and b are univariate polynomials over $\mathbb{Z}$ of degree less than $\deg h$, with coefficients bounded in absolute value by A. Up to a constant factor which can be assumed to be smooth without loss of generality, the norm $N_i(\phi)$ in the field defined by $f_i(X)$ is then given by $\operatorname{Res}_\iota\left(\operatorname{Res}_X\left(a(\iota) + b(\iota)X, f_i(X)\right), h(\iota)\right)$, and this can be bounded in absolute value by

$$|N_i(\phi)| \leq A^{(\deg h)(\deg f_i)} ||f_i||_\infty^{\deg h} ||h||_\infty^{\deg f_i((\deg h)-1)} C(\deg h, \deg f_i),$$

the combinatorial contribution C being $C(x, y) = (x+1)^{(3y+1)x/2}(y+1)^{3x/2}$.

In the case of TNFS where n is prime, the degree of h is equal to n, thus both factors of the combinatorial contribution are non-negligible. On the other hand, when $n = \eta\kappa$ is composite with appropriate factor sizes, one can use exTNFS and take $\deg h = \eta$ and $\deg f \geq \kappa$, in such a way that only the first factor of C will contribute in a non-negligible way to the size of the norm.

3.3.3 Methodology for the complexity analysis of NFS

During sieving, we explore A^t candidates, for which a smoothness test is performed. A single smoothness test with ECM has a cost that is non-polynomial. It is sub-exponential in the smoothness bound, meaning in $L_B(1/2)$. By the properties of the L_Q-notation, more precisely the fact that $L_{L_Q(\alpha)}(\beta) = L_Q(\alpha\beta)$, this gives an $L_Q(1/6)$ complexity and therefore contributes only in the $o(1)$ in the final complexity. We therefore count it as a unit cost in our analysis. In the plain NFS, the cost of sieving is therefore A^t. In the asymmetric MNFS, we should in principle add the cost of testing the smoothness of the $V - 1$ remaining norms when the first one is smooth. With the notations of Table 3.1, the sieving cost is therefore $A^t(1 + P_1 V)$. In what follows, we will assume that $P_1 V \ll 1$, i.e. $p_{r1} + c_V < 0$ and check at the end that this hypothesis is valid. As for the linear algebra, the cost is quadratic in the size of the factor basis. According to the prime number theorem, the number of prime ideals of norm bounded by B is proportional to B up to a logarithmic factor. In the asymmetric MNFS setting, the cost is $(B + VB')^2$, and in general, we balance the two terms and set $B = VB'$. Therefore, in the main case where we assume

equality between sieving and linear algebra, for both the plain NFS and the asymmetric MNFS variant we get $A^t = B^2$.

The Main Constraint also requires to have as many relations as the size of the factor bases. This translates into the equation $A^t P = B$, where P is the probability of finding a relation, which is equal to $P_1 P_2 V$ in the MNFS case. Combining this with the first constraint simplifies to $BP = 1$, or, in terms of exponents in the L_Q-notation:

$$p_r + c_B = 0, \tag{3.2}$$

where $p_r = p_{r_1} + p_{r_2} + c_V$ in the case of MNFS.

From the characteristics of the polynomials outputted by the polynomial selection, one can use the formulae of Section 3.3.2 to express p_r in terms of the parameters c_B, c_t, and also c_V in the MNFS case. Note that we use the equation $A^t = B^2$ to rewrite c_A as $c_A = 2c_B/c_t$.

It remains to find a minimum for the cost of the algorithm under the constraint given by Equation (3.2). To do so, we use Lagrange multipliers. Let $c = p_r + c_B$ be the constraint seen as a function of the continuous parameters. The Lagrangian function is given by $\mathcal{L}(\text{parameters}, \lambda) = 2c_B + \lambda c$, where λ is an additional non-zero variable. At a local minimum for the cost, all the partial derivatives of $\mathcal{L}$ are zero, and this gives a system of equations with as many equations as indeterminates (not counting c_p which is seen as a fixed parameter). Since all the equations are polynomials, it is then possible to use Gröbner basis techniques to express the minimum complexity as a function of c_p.

More precisely, in the case of the asymmetric MNFS where the variables are c_B, c_t and c_V, the system of equations is

$$\begin{cases} \frac{\partial \mathcal{L}}{\partial c_B} = 2 + \lambda \frac{\partial c}{\partial c_B} = 0 \\ \frac{\partial \mathcal{L}}{\partial c_t} = \lambda \frac{\partial c}{\partial c_t} = 0 \\ \frac{\partial \mathcal{L}}{\partial c_V} = \lambda \frac{\partial c}{\partial c_V} = 0 \\ p_{r1} + p_{r2} + c_V + c_B = 0 \, , \end{cases}$$

where the first equation plays no role in the resolution, but ensures that λ is non-zero, thus allowing to remove the λ in the second and third equation. This becomes an even simpler system in the case of the plain NFS, where the parameter c_V is no longer present, thus leading to the system

$$\begin{cases} \frac{\partial c}{\partial c_t} = 0 \\ p_r + c_B = 0 \, . \end{cases}$$

In the case where the expressions depend on discrete parameters, we can keep them in the formulae (without computing partial derivatives with respect to them, which would not make sense) and compute a parametrized Gröbner basis. If this leads to a system for which the Gröbner basis computation is too hard, then we can instantiate some or all the discrete parameters and then solve the system for each choice.

The cases not covered by the above setting, including the cases where we do not assume equality between sieving and linear algebra are handled similarly.

3.4 Polynomial selections

The asymptotic complexity of all the algorithms which were introduced in Chapter 1 depends on the characteristics of the polynomials outputted by the different polynomial selection methods. We briefly summarize in this section these various existing polynomial selection methods. We distinguish the cases when n is composite and when p is of a special form, which leads to considering different algorithms. The parameters for all the polynomial selections we are going to examine are summarized in Table 3.2.

3.4.1 Polynomial selections for NFS and MNFS

We first list the methods where no particular considerations are made on the extension degree n or the characteristic p.

JLSV0. This is the simplest polynomial method there exists. We consider a polynomial f_1 of degree n irreducible mod p such that the coefficients of f_1 are in $O(1)$. We construct $f_2 = f_1 + p$ and thus coefficients of f_2 are in $O(p)$ and the degree of f_2 is also n. Then trivially we have the condition that

$f_2|f_1 \mod p$ as required for the algorithms to work.

JLSV1. The JLSV1 method was introduced in [JLSV06]. We start by considering two polynomials h_0 and h_1 with small coefficients and such that $\deg h_0 = n$ and $\deg h_1 < n$. Consider some value $m > \sqrt{p}$ and let (u, v) be such that $m \equiv \frac{u}{v} \pmod{p}$. Then, define $f_1 = h_0 + mh_1$. The polynomial f_1 is then of degree n and its coefficients are of size $O(\sqrt{p})$. Secondly, define $f_2 = vh_0 + uh_1$. Similarly, the degree of f_2 is n and its coefficients are also of size $O(\sqrt{p})$. The previous steps are repeated until f_1 and f_2 are irreducible mod p. Both polynomials f_1, f_2 share a common factor   mod p. Note that the degree of the polynomials outputted are the same as those of JLSV0. However, the size of coefficients are balanced as opposed to JLSV0. This difference does not affect the overall complexity of the algorithm.

Polynomial selection algorithms such as JLSV2, GJL and $\mathcal{A}$ use lattices to output the second polynomial, the idea being that in order to produce a polynomial with small coefficients, the latter are chosen to be the coefficients of a short vector in a reduced lattice basis. We introduce the following lattice basis construction

$$M_k(p, f) = \left(\begin{array}{c|c} P & 0 \\ \hline F & 1 \end{array} \right)$$

where P is a diagonal $n \times n$ matrix with entries p, F is a $k - n + 1$-dimensional matrix whose rows corresponds to the coefficients of the polynomial f shifted by one index at each row and 1 simply corresponds to a diagonal matrix with 1 as entries.

JLSV2. This polynomial selection is presented in [JLSV06]. It constructs two irreducible polynomials f_1 and f_2 with integer coefficients such that f_1 is a degree n polynomial, irreducible modulo p, and f_2 is divisible by f_1 modulo p. This construction predates the Generalized Joux-Lercier method [BGGM15], one of the current polynomial selections used for large characteristic finite fields.

First we choose a monic irreducible polynomial h of degree n with small integer coefficients, say in $O(1)$. We select W an integer such that $f_1(x) = h(x + W)$ is irreducible modulo p. Since h is of degree n and monic, f_1 is also of degree n and monic. Besides, the largest coefficient of f_1 is $O(W^n)$. Let

$$f_1(x) = s_0 + s_1 x + \cdots + s_{n-1} x^{n-1} + x^n.$$

We look for a polynomial f_2 of degree d for some $d > n$, that is a multiple of f_1 modulo p and whose coefficients are smaller than W^n, i.e., smaller than those of f_1. To do so, we consider the lattice of the polynomials of degree at most d that are divisible by f_1 modulo p. This is a lattice of dimension $d + 1$ for which a basis is given by the rows of the matrix $M_d(p, f_1)$. The first n rows correspond to the polynomials $p, px, \cdots, px^{n-1}$ and the last $d - n + 1$ rows correspond to $f_1, xf_1, \cdots x^{d-n}f_1$. We can LLL-reduce this basis and expect to find vectors with small coefficients: We let f_2 be the polynomial which coefficients are given by the first vector in the reduced basis. Using LLL heuristic approximation factor, we expect that if $\mathbf{v}_1$ is the shortest vector in the lattice, then $||\mathbf{v}_1|| \approx (1.02)^{\dim M} \det(M)^{1/\dim M}$, where $||.||$ denotes the Euclidean norm. In our case, the determinant of M is p^n, and so $||f_2|| \approx (1.02)^{d+1} p^{n/(d+1)}$.

Note that the value $(1.02)^{d+1}$ is negligible here, so that, from a complexity analysis perspective, there is no need to use a better lattice reduction than LLL. Since $||f_1|| > W^n$, to ensure that f_2 is different from f_1, it is sufficient to choose W such that $||f_2|| \leq W^n$. Besides, for two given degrees we know that the lower the coefficients of the polynomials are, the faster NFS is. So to further optimize the asymptotic complexity we choose the smallest possible W, namely $W \approx p^{1/(d+1)}$. Finally, this polynomial selection outputs the polynomials f_1 and f_2. By construction, f_2 is divisible by f_1 mod p. The best value for d will be determined later on.

GJL. The Generalized Joux-Lercier (GJL) method is an extension to the non-prime fields of the method presented in 2003 by Joux and Lercier in [JL03]. It was proposed by Barbulescu, Gaudry, Guillevic and Morain in [BGGM15, Paragraph 6.2], and uses lattice reduction to build polynomials with small coefficients. We briefly recall the steps of this polynomial selection. First select an irreducible polynomial f_1 in $\mathbb{F}_p[X]$ of degree $d + 1 > n$ with coefficients in $O(\log p^n)$ and such that it has an irreducible monic

factor I of degree n modulo p. We can write $I(x) = x^n + \sum_{i=0}^{n-1} I_i x^i$. The polynomial selection needs to output a second polynomial f_2 which shares the same irreducible factor modulo p. To do so, we build the lattice of dimension $(d+1) \times (d+1)$ whose basis is given by $M_d(p, I - x^n)$.

Similarly to the JLSV2 construction, the polynomial f_2 is defined as a linear combination of polynomials of the form IX^k and pX^k. More precisely, its coefficients are the coefficients of the first vector in the LLL reduced basis of M. This allows to have smaller coefficients. The determinant of this lattice is p^n. Hence, running the LLL algorithm on M gives a polynomial of degree at most d that has coefficients of size at most $p^{n/d+1}$. Finally, we obtain two polynomials f_1 and f_2 that share a common degree n factor over $\mathbb{F}_p[X]$.

Conjugation. In [BGGM15], in addition to GJL, the authors propose another new polynomial selection method known as Conjugation. It uses a continued fraction method like JLSV1 and the existence of some square roots in $\mathbb{F}_p$. We first choose a quadratic monic polynomial ϕ with coefficients of size $O(\log p)$ which is irreducible over $\mathbb{Z}$ and has a root m in $\mathbb{F}_p$. We then consider two polynomials h_0 and h_1 with small coefficients and such that $\deg h_0 = n$ and $\deg h_1 < n$. Let (u, v) be such that $m \equiv u/v \mod p$. Then, define $f_2 = v h_0 + u h_1$. The polynomial f_2 is then of degree n and its coefficients are of size $O(\sqrt{p})$. Secondly, define $f_1 = \mathrm{Res}_y(\phi(y), h_0(x) + y h_1(x))$. Then, the degree of f_1 is $2n$ and its coefficients are of size $O(\log p)$. Both polynomials f_1, f_2 share a common factor mod p.

Algorithm $\mathcal{A}$. We recall Algorithm $\mathcal{A}$ as given in [SS16c] in Algorithm 6 as this variant leads to the best complexity at the boundary case. We refer to [SS16c] for more details about it. This algorithm also uses lattices to output the second polynomial and introduces two new parameters, $\hat{d}$ and r, such that $r \geq \frac{n}{\hat{d}} := k$. The parameters r and k are discrete in the complexity analyses. Note that the parameter $\hat{d}$ used in this polynomial selection is also discrete whereas the polynomial degree denoted d used in JLSV2 and GJL is continuous (we specify this here due to the very similar notation).

Algorithm 6 Algorithm $\mathcal{A}$

Input: p, n, $\hat{d}$, a factor of n and $r \geq n/\hat{d}$
Output: $f_1(x)$, $f_2(x)$ and $I(x)$
Let $k = n/\hat{d}$.

1: **repeat**

Randomly choose a monic irreducible polynomial $A_1(x)$ with the following properties: $\deg A_1(x) = r+1$, $A_1(x)$ is irreducible over the integers, $A_1(x)$ has coefficients of size $O(\log(p))$ and modulo p, $A_1(x)$ has an irreducible factor $A_2(x)$ of degree k.

Randomly choose monic polynomials $C_0(x)$ and $C_1(x)$ with small coefficients such that $\deg C_0(x) = \hat{d}$ and $\deg C_1(x) < \hat{d}$.

Define

$$f_1(x) = \mathrm{Res}_y(A_1(y), C_0(x) + y C_1(x))$$

$$I(x) = \mathrm{Res}_y(A_2(y), C_0(x) + y C_1(x)) \pmod{p}$$

$$J(x) = \mathrm{LLL}(M_r(p, A_2))$$

$$f_2(x) = \mathrm{Res}_y(J(y), C_0(x) + y C_1(x))$$

2: **until** $f_1(x)$ and $f_2(x)$ are irreducible over $\mathbb{Z}$ and $I(x)$ is irreducible over $\mathbb{F}_p$

return $f(x), g(x)$ and $I(x)$.

3.4.2 Polynomial selections for exTNFS and MexTNFS

We now look at polynomial selections with composite extension degree $n = \eta \kappa$. The most general algorithms are the algorithms $\mathcal{B}, \mathcal{C}$ and $\mathcal{D}$ presented in [SS16b, SS19] that extend algorithm $\mathcal{A}$ to the composite case. Thus, the construction of the polynomials f_1 and f_2 follow very similar steps as the ones in algorithm $\mathcal{A}$. We merely point out the main differences with algorithm $\mathcal{A}$. These algorithms require

the additional condition $\gcd(\eta, \kappa) = 1$. Similarly as for algorithm $\mathcal{A}$, they introduce two new parameters: $\hat{d}$ and r such that $r \geq \kappa/\hat{d} := k$.

Algorithm $\mathcal{B}$. This algorithm is identical to algorithm $\mathcal{A}$ adapted to the composite setup where $n = \eta\kappa$. Note that if $\eta = 1$ and $\kappa = n$, we recover algorithm $\mathcal{A}$. We recall it in Algorithm 7 as it leads to the best complexity at this boundary case for composite extension degrees.

Algorithm 7 Algorithm $\mathcal{B}$

    Input: p, $n = \kappa\eta$, $\hat{d}$, a factor of κ and $r \geq \kappa/\hat{d}$
    Output: $f_1(x)$, $f_2(x)$ and $I(x)$
Let $k = \kappa/\hat{d}$.

1: **repeat**
    Randomly choose a monic irreducible polynomial $A_1(x)$ with the following properties: $\deg A_1(x) = r + 1$, $A_1(x)$ is irreducible over the integers, $A_1(x)$ has coefficients of size $O(\log(p))$ and modulo p, $A_1(x)$ has an irreducible factor $A_2(x)$ of degree k.
    Randomly choose monic polynomials $C_0(x)$ and $C_1(x)$ with small coefficients such that $\deg C_0(x) = \hat{d}$ and $\deg C_1(x) < \hat{d}$.
    Define $\triangleright$ Notations are as in [SS16c]

$$f_1(x) = \mathrm{Res}_y(A_1(y), C_0(x) + yC_1(x))$$

$$I(x) = \mathrm{Res}_y(A_2(y), C_0(x) + yC_1(x)) \pmod{p}$$

$$J(x) = \mathrm{LLL}(M_r(p, A_2))$$

$$f_2(x) = \mathrm{Res}_y(J(y), C_0(x) + yC_1(x))$$

2: **until** $f_1(x)$ and $f_2(x)$ are irreducible over $\mathbb{Z}$ and $I(x)$ is irreducible over $\mathbb{F}_p$
    return $f_1(x)$, $f_2(x)$ and $I(x)$.

Algorithms $\mathcal{C}$ and $\mathcal{D}$. The polynomial selection $\mathcal{C}$ is another extension of $\mathcal{A}$ to the setup of exTNFS. It introduces a new variable $\lambda \in [1, \eta]$ that plays a crucial role in controlling the size of the coefficients of f_2. However, in our case, when analysing the complexity of M(ex)TNFS-$\mathcal{C}$ one realizes that the lowest complexity is achieved when $\lambda = 1$ which brings us back to the analysis of $\mathcal{B}$. As for algorithm $\mathcal{D}$, this is a variant that allows to replace the condition $\gcd(\eta, \kappa) = 1$ by the weaker condition $\gcd(\eta, k) = 1$. Since the outputted polynomials share again the same properties as algorithm $\mathcal{B}$, the complexity analysis is identical. Therefore, we will not consider $\mathcal{C}$ or $\mathcal{D}$ in the rest of the chapter.

3.4.3   Polynomial selections for SNFS and STNFS

For SNFS and STNFS, the prime p is given as the evaluation of a polynomial P of some degree λ and with small coefficients. In particular, we can write $p = P(u)$, for $u \approx p^{1/\lambda}$. Note that the degree λ is a fixed parameter which does not depend on p. We summarize the construction of the polynomials f_1 and f_2 given in [JP14b]. The first polynomial f_1 is defined as an irreducible polynomial over $\mathbb{F}_p$ of degree n and can be written as $f_1(x) = x^n + R(x) - u$, where R is a polynomial of small degree and coefficients taken in the set $\{-1, 0, 1\}$. The polynomial R does not depend on P so $||f_1||_\infty = u$, and from $p = P(u)$ we get $||f_1||_\infty = p^{1/\lambda}$. The polynomial f_2 is chosen to be $f_2(x) = P(f_1(x) + u)$. This implies $f_2(x) \pmod{f_1(x)} = p$, and thus $f_2(x)$ is a multiple of $f_1(x)$ modulo p. For STNFS it suffices to replace n by κ.

3.5   Complexity analyses of (M)(ex)(T)NFS

Following the method explained in Section 3.3, we have computed the complexities of the algorithms presented in Chapter 1, Section 1.3 with the polynomial selections given in Section 3.4. We report the norms and the complexities in Table 3.3 and Figure 3.6. Since each norm has the form $L_Q(2/3, c)$, and each complexity has the form $L_Q(1/3, c)$, we only report the values of c in the table. We illustrate

Polynomial selection	NFS				MNFS			
	$\deg f_1$	$\deg f_2$	$\|f_1\|_\infty$	$\|f_2\|_\infty$	$\deg f_1$	$\deg f_2$	$\|f_1\|_\infty$	$\|f_2\|_\infty$
JLSV0	n	n	$O(1)$	$O(p)$	$-$	$-$	$-$	$-$
JLSV1	n	n	$O(\sqrt{p})$	$O(\sqrt{p})$	n	n	$O(\sqrt{p})$	$O(\sqrt{V}\sqrt{p})$
JLSV2	n	$d>n$	$O(p^{n/(d+1)})$	$O(p^{n/(d+1)})$	n	$d>n$	$O(p^{n/(d+1)})$	$O(\sqrt{V}p^{n/(d+1)})$
GJL	$d+1>n$	d	$O(1)$	$O(p^{n/(d+1)})$	$d+1>n$	d	$O(1)$	$O(\sqrt{V}p^{n/(d+1)})$
Conjugation	$2n$	n	$O(\log p)$	$O(\sqrt{p})$	$2n$	n	$O(\log p)$	$O(\sqrt{V}\sqrt{p})$
$\mathcal{A}$	$d(r+1)$	dr	$O(\log p)$	$O(p^{n/d(r+1)})$	$d(r+1)$	dr	$O(\log p)$	$O(\sqrt{V}p^{n/d(r+1)})$

Polynomial selection	exTNFS				MexTNFS			
	$\deg f_1$	$\deg f_2$	$\|f_1\|_\infty$	$\|f_2\|_\infty$	$\deg f_1$	$\deg f_2$	$\|f_1\|_\infty$	$\|f_2\|_\infty$
JLSV2	κ	$d>\kappa$	$O(p^{\kappa/(d+1)})$	$O(p^{\kappa/(d+1)})$	κ	$d>\kappa$	$O(p^{\kappa/(d+1)})$	$O(\sqrt{V}p^{\kappa/(d+1)})$
$\mathcal{B}$	$d(r+1)$	dr	$O(\log p)$	$O(p^{k/(r+1)})$	$d(r+1)$	dr	$O(\log p)$	$O(\sqrt{V}p^{k/(r+1)})$

Polynomial selection	$\deg f_1$	$\deg f_2$	$\|f_1\|_\infty$	$\|f_2\|_\infty$
SNFS	n	$n\lambda$	$p^{1/\lambda}$	$O((\log n)^\lambda)$
STNFS	κ	$\kappa\lambda$	$p^{1/\lambda}$	$O\left((\log \kappa)^\lambda\right)$

Table 3.2: Parameters of the polynomials f_1, f_2 outputted by various polynomial selection methods for (M)NFS in the first table, (M)exTNFS in the second table and S(T)NFS in the third table.

our methodology by giving details of the computation of the complexity analysis of the best performing variants.

3.5.1 (M)NFS

In the case of (M)NFS, the best complexity is achieved by the MNFS asymmetric variant using the polynomial selection $\mathcal{A}$ and when equating the cost of sieving and linear algebra. The continuous parameters to consider in this case are B, A, t, V, and the discrete parameters are r and k.

The norms of the polynomials outputted by the polynomial selection $\mathcal{A}$ are bounded by

$$|N_1| < t^{\hat{d}(r+1)/2}(\hat{d}(r+1))^t (\log p)^t A^{\hat{d}(r+1)},$$

and

$$|N_2| < t^{\hat{d}r/2}(\hat{d}r)^t Q^{t/\hat{d}(r+1)} \sqrt{V}^t A^{\hat{d}r}.$$

Using Corollary 3, we compute the probabilities of smoothness for both norms. The constants in the L_Q notation for these probabilities are given by $p_{r1} = \frac{-1}{3c_B}\left(\frac{r+1}{6kc_p} + \frac{(r+1)c_A}{k}\right)$, and $p_{r2} = \frac{-1}{3(c_B-c_V)}\left(\frac{r}{6kc_p} + \frac{rc_A}{k} + \frac{kc_t}{r+1} + \frac{c_t c_V}{2c_p}\right)$. Using the condition $P = 1/B$ allows us to obtain a non-linear equation in the various parameters considered given by $p_{r1} + p_{r2} + c_V + c_B = 0$.

Recall that the overall complexity is $L_Q(1/3, 2c_B)$. In order to minimize $2c_B$ under the non-linear constraint given above, we use Lagrange multipliers and solve the system exhibited in Section 3.3 with Gröbner basis. This allows us to obtain an equation of degree 15 in c_B, degree 9 in c_p, and degrees 10 and 8 in r and k given in Figure 3.4. Recall that r and k are discrete values.

One can loop over the possible values of r, k and keep the values which give the smallest complexity. When $c_p \geq 1.5$, the optimal set of parameters is given by $(r, k) = (1, 1)$. When $1.2 \leq c_p \leq 1.4$, the values of (r, k) need to be increased to find a valid complexity. For $c_p \leq 1.1$, no values of (r, k) allows us to find a positive root for c_V, thus there is no valid parameters with this method.

The last step of our strategy consists in recursing on each plausible boundary of the subvariety of parameters. This case is already covered by the previous steps. Indeed, the only parameter where it makes sense to consider the boundary is V, and when the latter goes to zero, this means we are considering NFS again.

As a sanity check, one can look at what happens when the complexity approaches the medium characteristic case.

$$90699264c_B^{15}c_p^7r^6k^6 - 362797056c_B^{13}c_p^9r^4k^8 + 423263232c_B^{15}c_p^7r^5k^6 - 967458816c_B^{13}c_p^9r^3k^8 +$$
$$786060288c_B^{15}c_p^7r^4k^6 - 846526464c_B^{13}c_p^9r^2k^8 - 43670016c_B^{15}c_p^7r^7k^5 + 725594112c_B^{15}c_p^7r^3k^6 -$$
$$184757760c_B^{12}c_p^7r^6k^6 + 114213888c_B^{11}c_p^8r^5k^7 - 241864704c_B^{13}c_p^9rk^8 + 618098688c_B^{10}c_p^9r^4k^8 -$$
$$230107392c_B^{13}c_p^6r^6k^5 + 332563968c_B^{15}c_p^7r^2k^6 - 765904896c_B^{12}c_p^7r^5k^6 + 369515520c_B^{11}c_p^8r^4k^7 +$$
$$1303382016c_B^{10}c_p^9r^3k^8 - 498845952c_B^{13}c_p^6r^5k^5 + 60466176c_B^{15}c_p^7rk^6 - 1242915840c_B^{12}c_p^7r^4k^6 +$$
$$429981696c_B^{11}c_p^8r^3k^7 + 846526464c_B^{10}c_p^9r^2k^8 + 1337408c_B^{11}c_p^5r^8k^4 - 566030592c_B^{13}c_p^6r^4k^5 +$$
$$30233088c_B^{10}c_p^6r^7k^5 - 977536512c_B^{12}c_p^7r^3k^6 + 16236288c_B^9c_p^7r^6k^6 + 201553920c_B^{11}c_p^8r^2k^7 +$$
$$7838208c_B^8c_p^8r^5k^7 + 147806208c_B^{10}c_p^9rk^8 + 10077696c_B^7c_p^9r^4k^8 + 63452160c_B^{11}c_p^5r^7k^4 -$$
$$349360128c_B^{13}c_p^6r^3k^5 + 140714496c_B^{10}c_p^6r^6k^5 - 362797056c_B^{12}c_p^7r^2k^6 + 66624768c_B^9c_p^7r^5k^6 +$$
$$20155392c_B^{11}c_p^8rk^7 + 41430528c_B^8c_p^8r^4k^7 - 13436928c_B^{10}c_p^9k^8 + 40310784c_B^7c_p^9r^3k^8 +$$
$$148599360c_B^{11}c_p^5r^6k^4 - 105815808c_B^{13}c_p^6r^2k^5 + 261180288c_B^{10}c_p^6r^5k^5 - 43670016c_B^{12}c_p^7rk^6 +$$
$$103389696c_B^9c_p^7r^4k^6 - 6718464c_B^{11}c_p^8k^7 + 85100544c_B^8c_p^8r^3k^7 + 60466176c_B^7c_p^9r^2k^8 +$$
$$82944c_B^9c_p^4r^9k^3 + 186577344c_B^{11}c_p^5r^5k^4 + 492480c_B^8c_p^5r^8k^4 - 8398080c_B^{13}c_p^6rk^5 +$$
$$237852288c_B^{10}c_p^6r^4k^5 + 720576c_B^7c_p^6r^7k^5 + 3359232c_B^{12}c_p^7k^6 + 71290368c_B^9c_p^7r^3k^6 +$$
$$311040c_B^6c_p^7r^6k^6 + 85100544c_B^8c_p^8r^2k^7 + 40310784c_B^7c_p^9rk^8 + 870912c_B^8c_p^5r^8k^3 +$$
$$132013152c_B^{11}c_p^5r^4k^4 + 3955392c_B^8c_p^5r^7k^4 + 1679616c_B^{13}c_p^6k^5 + 97417728c_B^{10}c_p^6r^3k^5 +$$
$$4950720c_B^7c_p^6r^6k^5 + 15863040c_B^9c_p^7r^2k^6 + 1897344c_B^6c_p^7r^5k^6 + 41430528c_B^8c_p^8rk^7 +$$
$$10077696c_B^7c_p^9k^8 + 3710448c_B^9c_p^4r^7k^3 + 48335616c_B^{11}c_p^5r^3k^4 + 13618368c_B^8c_p^5r^6k^4 +$$
$$1119744c_B^{10}c_p^6r^2k^5 + 14530752c_B^7c_p^6r^5k^5 - 4292352c_B^9c_p^7rk^6 + 4821120c_B^6c_p^7r^4k^6 +$$
$$7838208c_B^8c_p^8k^7 + 342c_B^7c_p^3r^{10}k^2 + 8715600c_B^9c_p^4r^6k^3 + 864c_B^6c_p^4r^9k^3 + 4758912c_B^{11}c_p^5r^2k^4 +$$
$$26326944c_B^8c_p^5r^5k^4 + 648c_B^5c_p^5r^8k^4 - 11850624c_B^{10}c_p^6rk^5 + 23602752c_B^7c_p^6r^4k^5 +$$
$$288c_B^4c_p^6r^7k^5 - 1866240c_B^9c_p^7k^6 + 6531840c_B^6c_p^7r^3k^6 + 3204c_B^7c_p^3r^9k^2 +$$
$$12650256c_B^9c_p^4r^5k^3 + 8352c_B^6c_p^4r^8k^3 - 2192832c_B^{11}c_p^5rk^4 + 31313952c_B^8c_p^5r^4k^4 +$$
$$6192c_B^5c_p^5r^7k^4 - 2706048c_B^{10}c_p^6k^5 + 22897728c_B^7c_p^6r^3k^5 + 2016c_B^4c_p^6r^6k^5 +$$
$$4976640c_B^6c_p^7r^2k^6 + 14112c_B^7c_p^3r^8k^2 + 11875248c_B^9c_p^4r^4k^3 + 35928c_B^6c_p^4r^7k^3 -$$
$$536544c_B^{11}c_p^5k^4 + 23493888c_B^8c_p^5r^3k^4 + 25128c_B^5c_p^5r^6k^4 + 13255488c_B^7c_p^6r^2k^5 +$$
$$6048c_B^4c_p^6r^5k^5 + 2021760c_B^6c_p^7rk^6 + 2c_B^5c_p^2r^{11}k + 37980c_B^7c_p^3r^7k^2 + 10c_B^4c_p^3r^{10}k^2 +$$
$$7248528c_B^9c_p^4r^3k^3 + 89928c_B^6c_p^4r^6k^3 + 8c_B^3c_p^4r^9k^3 + 10865664c_B^8c_p^5r^2k^4 +$$
$$57024c_B^5c_p^5r^5k^4 + 4235328c_B^7c_p^6rk^5 + 10080c_B^4c_p^6r^4k^5 + 342144c_B^6c_p^7k^6 + 25c_B^5c_p^2r^{10}k +$$
$$68184c_B^7c_p^3r^6k^2 + 100c_B^4c_p^3r^9k^2 + 2782512c_B^9c_p^4r^2k^3 + 143928c_B^6c_p^4r^5k^3 + 72c_B^3c_p^4r^8k^3 +$$
$$2833056c_B^8c_p^5rk^4 + 79560c_B^5c_p^5r^4k^4 + 575424c_B^7c_p^6k^5 + 10080c_B^4c_p^6r^3k^5 + 136c_B^5c_p^2r^9k +$$
$$84276c_B^7c_p^3r^5k^2 + 448c_B^4c_p^3r^8k^2 + 610416c_B^9c_p^4rk^3 + 152424c_B^6c_p^4r^4k^3 + 288c_B^3c_p^4r^7k^3 +$$
$$318816c_B^8c_p^5k^4 + 70128c_B^5c_p^5r^3k^4 + 6048c_B^4c_p^6r^2k^5 + 430c_B^5c_p^2r^8k + 71964c_B^7c_p^3r^4k^2 +$$
$$1184c_B^4c_p^3r^7k^2 + 58320c_B^9c_p^4k^3 + 106632c_B^6c_p^4r^3k^3 + 672c_B^3c_p^4r^6k^3 + 38232c_B^5c_p^5r^2k^4 +$$
$$2016c_B^4c_p^6rk^5 + 884c_B^5c_p^2r^7k + 41652c_B^7c_p^3r^3k^2 + 2044c_B^4c_p^3r^6k^2 + 47448c_B^6c_p^4r^2k^3 +$$
$$1008c_B^3c_p^4r^5k^3 + 11808c_B^5c_p^5rk^4 + 288c_B^4c_p^6k^5 + 1246c_B^5c_p^2r^6k + 15570c_B^7c_p^3r^2k^2 +$$
$$2408c_B^4c_p^3r^5k^2 + 12168c_B^6c_p^4rk^3 + 1008c_B^3c_p^4r^4k^3 + 1584c_B^5c_p^5k^4 + 1232c_B^5c_p^2r^5k +$$
$$3384c_B^7c_p^3rk^2 + 1960c_B^4c_p^3r^4k^2 + 1368c_B^6c_p^4k^3 + 672c_B^3c_p^4r^3k^3 + 856c_B^5c_p^2r^4k +$$
$$324c_B^7c_p^3k^2 + 1088c_B^4c_p^3r^3k^2 + 288c_B^3c_p^4r^2k^3 + 410c_B^5c_p^2r^3k + 394c_B^4c_p^3r^2k^2 + 72c_B^3c_p^4rk^3 +$$
$$129c_B^5c_p^2r^2k + 84c_B^4c_p^3rk^2 + 8c_B^3c_p^4k^3 + 24c_B^5c_p^2rk + 8c_B^4c_p^3k^2 + 2c_B^5c_p^2k = 0$$

Figure 3.4: Equation for MNFS-A: (the parameters r and k are discrete)

Question 35. *What happens when $c_p \to \infty$?*

When the value of c_p tends to infinity, we expect to recover the complexity of MNFS-$\mathcal{A}$ in the medium characteristic case. Indeed, when setting $r = k = 1$, the equation resulting from solving the Lagrange multipliers-system becomes

$$-15c_B^6 + 18c_B^3 + 1 = 0,$$

which admits a unique positive real root $c_B = \sqrt[3]{\frac{3}{5} + \frac{4\sqrt{\frac{2}{3}}}{5}}$. Finally the asymptotic complexity of MNFS-$\mathcal{A}$ is given by $L_Q\left(1/3, 2c_B\right)$, with $2c_B = 2\sqrt[3]{\frac{3}{5} + \frac{4\sqrt{\frac{2}{3}}}{5}} \approx 2.156$. This is exactly the value found in medium characteristic.

An attempt at lowering the complexity of MNFS. Some polynomial selections such as $\mathcal{A}$ and JLSV2 output two polynomials f_1 and f_2 where f_2 is taken to be the polynomial whose coefficients are the coefficients of the shortest vector in an LLL-reduced lattice of some dimension D. The remaining $V - 2$ number fields are defined by polynomials which are linear combinations of f_1 and f_2. From the properties of LLL, we assume the vectors in the LLL-reduced basis have similar norms. Instead of building f_i as $\alpha_i f_1 + \beta_i f_2$ where $\alpha_i, \beta_i \approx \sqrt{V}$, one can take a linear combination of more short vectors, and thus have $f_i = \alpha_{i,1} f_1 + \alpha_{i,2} f_2 + \cdots + \alpha_{i,D} f_D$ and $\alpha_{i,j} \approx V^{1/2D}$. However, this does not affect the asymptotic complexity. When $c_p \to \infty$, the coefficient term becomes negligible. On the other hand, when c_p is small, the norms become smaller and this results in a slightly lower complexity. However the gain is very small, nearly negligible.

When looking at TNFS with n prime. We consider a linear polynomial g and a polynomial f of degree d where both polynomials have coefficients of size $O\left(p^{1/(d+1)}\right)$. This corresponds to the naive base-m polynomial selection. The TNFS setup requires a polynomial h of degree n with coefficients of size $O(1)$. As usual, to compute the complexity, we are interested in the size of the norms. This is given in Section 3.3.2 and when evaluating the term $C(n, d)$, which is not negligible due to the size of n as opposed to the large characteristic case presented in [BGK15], we note that the overall complexity of TNFS at this boundary case is greater than the usual $L_Q\left(1/3\right)$. Indeed, we have

$$\log C(n, d) = \frac{\delta}{c_p}(\log Q)^{4/3}(\log\log Q)^{-1/3} + \frac{4}{3c_p}(\log Q)^{2/3}(\log\log Q)^{1/3}.$$

Since $(\log Q)^{4/3}(\log\log Q)^{-1/3} > (\log Q)^{2/3}(\log\log Q)^{1/3}$ for large enough value of Q, we have $C(n, d) > L_Q(2/3, x)$ for any constant $x > 0$. Thus this algorithm is not applicable in this case. Moreover, if we write $p = L_Q\left(\alpha, c\right)$, this argument is valid as soon as $\alpha \leq 2/3$.

In order to apply the ideas of TNFS to the medium and small characteristic cases, we consider n to be composite and write $n = \eta\kappa$. This leads to the variant called exTNFS and its multiple number field extension, MexTNFS.

3.5.2 (M)exTNFS

When the extension degree $n = \eta\kappa$ is composite, using the extended TNFS algorithm and its multiple field variant allows to lower the overall complexity.

Before starting the complexity analysis, we want to underline a main difference with other analyses seen previously. So far, the degree t of the sieving polynomials has always been taken to be a function of $\log Q$, i.e., we usually set $t = \frac{c_t}{c_p}\left(\frac{\log Q}{\log\log Q}\right)^{1/3}$. In the following analysis, the value of t is a discrete value. Indeed, if one chooses to analyze the complexity using t as a function of $\log Q$, we get the following value in the product of the norms: $Q^{(t-1)/(d(r+1))} = L_Q\left(1, kc_t c_\eta/(r+1)\right)$. This implies that the norms become too big to give a final complexity in $L_Q\left(1/3\right)$.

We now concentrate on the analysis of exTNFS, using Algorithm $\mathcal{B}$. Continuous parameters are B, A, η and the discrete values are r, k, t. For simplicity we report only the case $t = 2$. The product of the

norms is bounded by

$$|N_1 N_2| < A^{\eta \hat{d}(2r+1)} p^{k\eta/(r+1)} C(\eta, \hat{d}r) C(\eta, \hat{d}(r+1)).$$

The two combinatorial terms are not negligible at this boundary case. The probability of getting relations is thus given by

$$P = L_Q \left(\frac{1}{3}, \frac{-1}{3c_B} \left(\frac{(2r+1)c_A}{k} + \frac{kc_\eta c_p}{r+1} + \frac{2r+1}{2kc_p} \right) \right),$$

and using the condition $P = 1/B$ allows us to obtain a non-linear equation in the various parameters considered given by $p_r + c_B = 0$.

In order to minimize $2c_B$ under this non-linear constraint, we proceed as before and use Lagrange multipliers and solve the system exhibited in Section 3.3 with a Gröbner basis approach. This allows us to obtain an equation of degree 4 in c_B and r and degree 2 in c_p and k. The equation is given below where the parameters r, k are discrete.

$$36c_p^2 c_B^4 r^2 k^2 + 72c_p^2 c_B^4 rk^2 + 36c_p^2 c_B^4 k^2 - 24c_p c_B^2 r^3 k - 32c_p^2 c_B r^2 k^2 - 60c_p c_B^2 r^2 k -$$
$$48c_p^2 c_B rk^2 - 48c_p c_B^2 rk - 16c_p^2 c_B k^2 + 4r^4 - 12c_p c_B^2 k + 12r^3 + 13r^2 + 6r + 1 = 0$$

Since r, k are discrete values, one can then loop through their possible values and pick the ones which give the smallest complexity.

Again, we can test the complexity when approaching the medium characteristic case.

Question 36. *What happens when $c_p \to \infty$?*

When $c_p \to \infty$, the optimal complexity is achieved when $r = k = 1$ and the above equation becomes

$$144c_B^4 c_p^2 - 144c_B^2 c_p - 96c_B c_p^2 + 36 = 0.$$

Hence, when $c_p \to \infty$, the complexity converges to $2c_B = \left(\frac{48}{9} \right)^{1/3}$ which is the complexity found in medium characteristics for exTNFS.

Question 37. *What happens when one adds the multiple variant to the extended tower variant?*

In the context of MexTNFS, the constants in the L_Q-notation for the probabilities become

$$p_{r1} = \frac{-1}{3c_B} \left(\frac{c_A(r+1)}{k} + \frac{r+1}{2kc_p} \right),$$

and

$$p_{r2} = \frac{-1}{3(c_B - c_V)} \left(\frac{rc_A}{k} + \frac{kc_p c_\eta}{r+1} + \frac{c_V c_\eta}{2} + \frac{r}{2kc_p} \right).$$

We proceed as usual by considering the condition $P = 1/B$ which gives a non-linear constraint and we minimize $2c_B$ under this constraint. We obtain an equation of degree 15 in c_B, of degree 8 in c_p and of degrees 11 and 8 in r and k (which we do not provide this time). Again, since r and k are discrete values, one can loop through their possible values to minimize the complexity. As c_p increases, the multiple variant allows to decrease the complexity, as seen in Figure 3.5.

Again, we can test the complexity when approaching the medium characteristic case.

Question 38. *What happens when $c_p \to \infty$?*

When $r = k = 1$, the equation mentioned above becomes

$$-29859840c_B^{15}c_p^6 + 29859840c_B^{13}c_p^8 + 48439296c_B^{13}c_p^5 + 30357504c_B^{12}c_p^6 - 31850496c_B^{11}c_p^7 - 17915904c_B^{10}c_p^8$$
$$-24868224c_B^{11}c_p^4 - 22118400c_B^{10}c_p^5 + 857088c_B^9 c_p^6 - 497664c_B^7 c_p^8 + 3866112c_B^9 c_p^3 + 2112000c_B^8 c_p^4$$
$$-751616c_B^7 c_p^5 - 55296c_B^6 c_p^6 - 237312c_B^7 c_p^2 - 84480c_B^6 c_p^3 + 52736c_B^5 c_p^4 - 6144c_B^4 c_p^5 + 6912c_B^5 c_p$$
$$+1536c_B^4 c_p^2 - 2048c_B^3 c_p^3 = 0$$

and when $c_p \to \infty$, this equation admits for unique real positive root $c_B = \sqrt[3]{\frac{3}{10} + \frac{2\sqrt{\frac{2}{3}}}{5}}$. The final asymptotic complexity is then given by $2c_B \approx 1.71$ and we recover the value given in [SS16b] in the medium characteristic case for MexTNFS. This value is as expected slightly lower than the asymptotic complexity of exTNFS which is equal to $\left(\frac{48}{9}\right)^{1/3} \approx 1.74$.
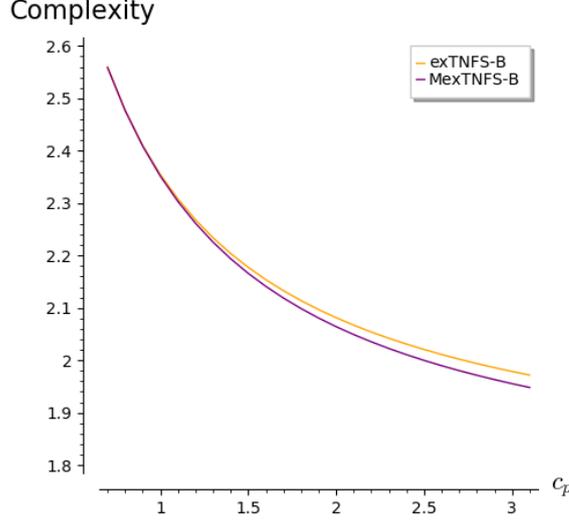
Figure 3.5: Comparing MexTNFS and exTNFS with polynomial selection $\mathcal{B}$. The variant exTNFS converges to ≈ 1.74 whereas MexTNFS converges to ≈ 1.71 when $c_p \to \infty$.

A note on the JLSV2 polynomial selection. When considering the JLSV2 polynomial selection for exTNFS (same for MexTNFS), the norms are bounded by

$$|N_1| < A^{\eta\kappa}||f||_\infty^\eta C(\eta, \kappa) = A^{\eta\kappa} p^{\kappa\eta/(d+1)} C(\eta, \kappa),$$
$$|N_2| < A^{\eta d}||g||_\infty^\eta C(\eta, d) = A^{\eta d} p^{\kappa\eta/(d+1)} C(\eta, d).$$

The terms $C(\eta, \kappa)$ and $C(\eta, d)$ are not negligible at this boundary case, and $C(\eta, \kappa) = L_Q(2/3, c_\eta c_\kappa/2)$. Similarly, we have $C(\eta, d) = L_Q(2/3, \delta c_\eta/2)$. By looking at the terms in N_2 and the value of $C(\eta, d)$, one notes that the norm is minimized when $\eta = 1$ since it only appears in the numerators. This means that n is not composite. Thus, no improvement to JLSV2 can be obtained by considering a composite n.

3.5.3 S(T)NFS

We give as an example the complexity analysis of SNFS and then explain why STNFS is not applicable at this boundary case.

SNFS. From the characteristics of the polynomials outputted by the polynomial selection used for SNFS given in Table 3.2, we compute the product of the norms which is given by

$$|N_1 N_2| < n^{2t} \lambda^t t^{n(\lambda+1)} p^{1/\lambda} A^{n(\lambda+1)} (\log(n))^{\lambda t}.$$

The probability that both norms are smooth is given by $P = L_Q\left(\frac{1}{3}, \frac{-1}{3c_B}\left(\frac{\lambda+1}{3c_p} + (\lambda+1)c_A + \frac{c_t}{\lambda}\right)\right)$. We consider the usual constraint given by the NFS analysis, $c_B + p = 0$. By deriving this constraint with respect to c_t and using a Gröbner basis approach, we obtain the following equation of c_B as a function of c_p:

$$81 c_B^4 c_p^2 \lambda^2 - 18 c_B^2 c_p \lambda^3 - 18 c_B^2 c_p \lambda^2 - 72 c_B c_p^2 \lambda^2 - 72 c_B c_p^2 \lambda + \lambda^4 + 2\lambda^3 + \lambda^2 = 0.$$

The complexities of SNFS with varying λ values can be seen in Figure 3.6. As done previously, we can test the complexity when approaching the medium characteristic case.

Algorithm	N_1	N_2	Complexity $2c_B$		
			$c_p = 1$	$c_p = 5$	$c_p \to \infty$
NFS-JLSV0	$\frac{1}{6c_p} + \frac{c_t}{2} + c_A$	$\frac{1}{6c_p} + \frac{c_t}{2} + c_A$	2.54	2.45	$\left(\frac{128}{9}\right)^{1/3} \approx 2.4$
NFS-JLSV1	$\frac{1}{6c_p} + \frac{c_t}{2} + c_A$	$\frac{1}{6c_p} + \frac{c_t}{2} + c_A$	2.54	2.45	$\left(\frac{128}{9}\right)^{1/3} \approx 2.4$
NFS-JLSV2	$\frac{1}{6c_p} + \frac{c_t}{\delta c_p} + c_A$	$\frac{\delta}{6} + \frac{c_t}{\delta c_p} + \delta c_A c_p$	2.87	2.62	$\left(\frac{128}{9}\right)^{1/3} \approx 2.4$
NFS-A	$\frac{r+1}{6kc_p} + \frac{(r+1)c_A}{k}$	$\frac{r}{6kc_p} + \frac{rc_A}{k} + \frac{kc_t}{r+1}$	2.39	2.24	$\left(\frac{96}{9}\right)^{1/3} \approx 2.2$
MNFS-JLSV1	$\frac{1}{6c_p} + \frac{c_t}{2} + c_A$	$\frac{1}{6c_p} + \frac{c_t}{2} + c_A + \frac{c_t c_V}{2c_p}$	2.52	2.36	$\frac{2\sqrt[3]{7+4\sqrt{3}}}{3^{2/3}} \approx 2.31$
MNFS-JLSV2	$\frac{1}{6c_p} + \frac{c_t}{\delta c_p} + c_A$	$\frac{\delta}{6} + \frac{c_t}{\delta c_p} + \delta c_p c_A + \frac{c_t c_V}{2c_p}$	–	2.62	$\frac{2}{3}\sqrt[3]{23 + \frac{13\sqrt{13}}{2}} \approx 2.396$
MNFS-A	$\frac{r+1}{6kc_p} + \frac{(r+1)c_A}{k}$	$\frac{r}{6kc_p} + \frac{rc_A}{k} + \frac{kc_t}{r+1} + \frac{c_t c_V}{2c_p}$	–	2.22	$2\sqrt[3]{\frac{3}{5} + \frac{4\sqrt{\frac{2}{3}}}{5}} \approx 2.156$
exTNFS-B	$\frac{(r+1)c_A}{k} + \frac{r+1}{2kc_p}$	$\frac{rc_A}{k} + \frac{kc_\eta c_p}{r+1} + \frac{r}{2kc_p}$	2.35	1.89	$\left(\frac{48}{9}\right)^{1/3} \approx 1.747$
MexTNFS-B	$\frac{(r+1)c_A}{k} + \frac{r+1}{2kc_p}$	$\frac{rc_A}{k} + \frac{kc_\eta c_p}{r+1} + \frac{r}{2kc_p} + \frac{c_V c_\eta}{2}$	2.35	1.86	$2\sqrt[3]{\frac{3}{10} + \frac{2\sqrt{\frac{2}{3}}}{5}} \approx 1.71$
SNFS-λ	$\frac{1}{6c_p} + \frac{c_t}{\lambda} + c_A$	$\frac{\lambda}{6c_p} + \lambda c_A$	–	–	$\left(\frac{64(\lambda+1)}{9\lambda}\right)^{1/3}$
SNFS-2	$\frac{1}{6c_p} + \frac{c_t}{2} + c_A$	$\frac{2}{6c_p} + 2c_A$	2.39	2.24	$\left(\frac{192}{18}\right)^{1/3} \approx 2.20$
SNFS-56	$\frac{1}{6c_p} + \frac{c_t}{56} + c_A$	$\frac{56}{6c_p} + 56c_A$	4.27	2.63	$\left(\frac{3648}{504}\right)^{1/3} \approx 1.93$
STNFS	$c_A + \frac{c_\eta c_p}{\lambda} + \frac{c_\eta c_\kappa}{2}$	$\lambda c_A + \frac{c_\eta c_\kappa \lambda}{2}$	–	–	–

Table 3.3: Norms and complexities for (M)(ex)(S)NFS algorithms.

> **Question 39.** *What happens when $c_p \to \infty$?*

When $c_p \to \infty$, the complexity is given by $2c_B = (64(\lambda + 1)/(9\lambda))^{1/3}$. When $\lambda = 1$, this value is equal to $(128/9)^{1/3}$. This is not surprising as λ is the degree of the polynomial P that defines p, and if $\lambda = \deg P = 1$, we are simply considering NFS. When $\lambda \geq 2$, the complexity becomes better than $(128/9)^{1/3}$. If λ is chosen to be a function of $\log Q$, for example if $\lambda = n$, then the norms become too big, and the resulting complexity is much higher.

STNFS. We look at the composite case where $n = \eta\kappa$ and consider the exTNFS algorithm with the special variant. From Table 3.2, we have the following norms: $|N_1| < A^n p^{\eta/\lambda} C(\eta, \kappa)$ and $|N_2| < A^{n\lambda}(\log \kappa)^{n\lambda} C(\eta, \kappa\lambda)$.

First, the term $(\log \kappa)^{n\lambda}$ is negligible due to the size of κ and η. Among the remaining terms, for a fixed λ value, one can see that the size of the norms is minimized when $\eta = 1$, thus when n is not composite. Hence, applying the special variant to the exTNFS algorithm will output a complexity greater than $L_Q(1/3)$ as already seen above. The STNFS algorithm can be used in medium characteristics for composite n as shown in [KB16]. In this case, the value of λ is chosen to be a function of $\log Q$, and allows to obtain a minimal value for the complexity where the value of η is not necessarily equal to 1. In particular, the product $n\lambda$ can be chosen such as to keep the norm in $L_Q(2/3)$ since n is not fixed as opposed to the boundary case.

Remark 7. *The algorithm MNFS-JLSV2 is absent from the left part of Figure 3.6 since for small values of c_p, the optimal value for the parameter V reaches its boundary and thus MNFS simply becomes NFS. This is also the case for MexTNFS-B and MNFS-JLSV1 for smaller values of c_p.*

3.6 Crossover points between NFS, FFS and the Quasi-Polynomial algorithms

Because so many algorithms cohabitate at this boundary case, in order to find out which one performs best, it is important to determine the exact crossover points between their complexities. In this section,
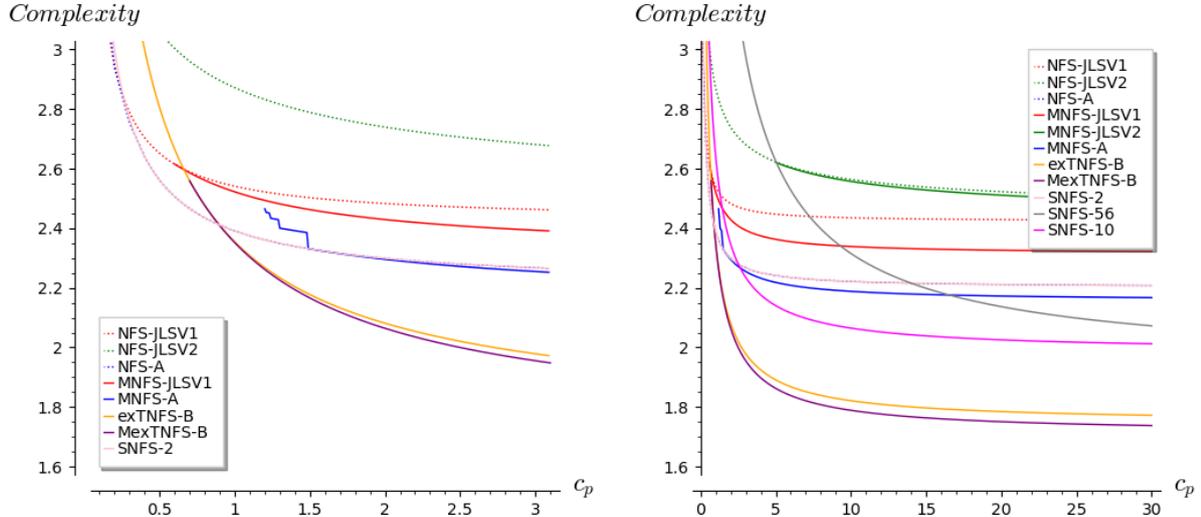
Figure 3.6: Complexities of NFS and all its variants as a function of c_p.

after briefly recalling the proven complexity of the quasi polynomial algorithms, we provide exact crossover points between FFS and QP and then between FFS and NFS (and its variants).

3.6.1 Quasi-Polynomial algorithms

After half a decade of both practical and theoretical improvements led by several teams and authors, the following result was finally proven in 2019:

Theorem 8 (Theorem 1.1. [KW19]). *Given any prime number p and any positive integer n, the discrete logarithm problem in the group $\mathbb{F}_{p^n}^{\times}$ can be solved in expected time $C_{QP} = (pn)^{2\log_2(n)+O(1)}$.*

This complexity is Quasi-Polynomial only when p is fixed or slowly grows with Q. When p is in the whereabouts of $L_Q(1/3)$ and n in $(\log Q)^{2/3}$, we obtain a complexity comparable to $L_Q(1/3)$. Therefore this algorithm must come into play in our study; we abbreviate it by QP, even if in our range of study its complexity is no longer Quasi-Polynomial.

3.6.2 Crossover between FFS and QP

When $p = L_Q\left(1/3, c_p\right)$, the complexity of QP algorithms is a power of the term $\exp\left(\log(Q)^{1/3}(\log\log Q)^{5/3}\right)$ larger than any $L_Q(1/3)$ expression. The crossover point is therefore for a characteristic p growing slower than an $L_Q(1/3)$ expression. In this area, the complexity of FFS is $C_{\text{FFS}} = L_Q(1/3, (32/9)^{1/3})$ or $C_{\text{shifted FFS}} = L_Q(1/3, (128/81)^{1/3})$ if n is composite and has a factor of exactly the right size so that the shifted FFS yields an optimal complexity. These complexities correspond to the complexities of (shifted) FFS when $c_p \to 0$.

The crossover point is when the expression of C_{QP} takes the $L_Q(1/3)$ form. More precisely, this occurs when p has the following expression

$$p = \exp\left(\gamma_p(\log Q)^{1/3}(\log\log Q)^{-1/3}\right) =: M_Q\left(1/3, \gamma_p\right),$$

where we define the notation $M_Q\left(\alpha, \beta\right) = \exp\left(\beta(\log Q)^{\alpha}(\log\log Q)^{-\alpha}\right)$. This M_Q function fits as follows with the L_Q function: for any positive constants α, β, γ, and ε, when Q grows to infinity we have the following inequalities $L_Q\left(1/3 - \varepsilon, \beta\right) \ll M_Q\left(1/3, \gamma\right) \ll L_Q\left(1/3, \alpha\right)$.

Writing $Q = p^n$ with p of this form, the formula for the extension degree n becomes

$$n = \frac{1}{\gamma_p}(\log Q)^{2/3}(\log\log Q)^{1/3},$$

so that the cost of the QP algorithm is

$$C_{\mathrm{QP}} = L_Q \left(\frac{1}{3}, \frac{4\gamma_p}{3\log 2} \right).$$

Equating this cost with the complexity of FFS, we obtain the crossover point. If only the non-shifted FFS is available, for instance because n is prime, then the crossover is when $p = M_Q \left(1/3, \left(\frac{3}{2}\right)^{1/3} \log 2 \right)$. Otherwise, if n has a factor of an appropriate size for the shifted FFS, the crossover is at the value $p = M_Q \left(1/3, \left(\frac{2}{3}\right)^{1/3} \log 2 \right)$.

3.6.3 Crossover between NFS and FFS

We compare the performance of FFS with the best variants of NFS. All complexities are expressed as $L_Q\,(1/3, c)$, where c is a function of c_p. Thus, it is enough to compare the values of c for each algorithm. Let c_{FFS} be this value in the case of FFS and c_{NFS} for NFS and all its variants.

We look for the value of c_p for which $c_{\mathrm{FFS}} = c_{\mathrm{NFS}}$, where the best variant of NFS depends on the considerations made on n and p. Indeed, when no special considerations are made on either n or p, the best algorithm among the variants of NFS is MNFS-$\mathcal{A}$ as seen in Section 8.4. When n is composite, the algorithm that performs best when c_p is small is (M)exTNFS-$\mathcal{B}$ depending on c_p. Finally, when p is taken to have a special form, the SNFS algorithm gives a good complexity when c_p is small and MNFS does not. For each of these algorithms, we know c_{NFS} as a function of c_p. Moreover, when looking at the FFS algorithm, we note that the crossover value is located in the area where the linear algebra phase is the dominant and that in this area the value of D is 1. Thus $c_{\mathrm{FFS}} = 2c_p$. Hence, we are able to compute exact values of these crossover points which we report in Table 3.4. The complexity of SNFS depends on the value of λ. We report in Table 3.4 the crossover points for $\lambda = 3, 20$ and 56. The smallest c_p value for the crossover point with FFS corresponds to $\lambda = 3$. Note also that for the range of c_p for which the NFS variants intersect FFS, the variant MexTNFS performs very similarly than exTNFS, and thus we only report the crossover point with MexTNFS.

	normal p	special p, $\lambda = 3, 20, 56$
n prime	$c_p = 1.23$, $c = 2.46$, MNFS-$\mathcal{A}$	$c_p = 1.17, 1.41, 1.75$, $c = 2.34, 2.81, 3.50$, SNFS-λ
n composite	$c_p = 1.14$, $c = 2.28$, MexTNFS-$\mathcal{B}$	

Table 3.4: Values of c_p for crossover points between FFS and variants of NFS, together with their relative complexities $L_Q(1/3, c)$.

3.7 Considering pairings

When constructing a pairing $e : \mathcal{E} \times \mathcal{E} \to \mathbb{F}_{p^n}$ for some elliptic curve $\mathcal{E}$ over the finite field $\mathbb{F}_p$, one must take into account the hardness of DLP in both a subgroup of $\mathcal{E}$ and in $\mathbb{F}_{p^n} = \mathbb{F}_Q$. A natural question arises.

> **Question 40.** *Asymptotically what finite field $\mathbb{F}_{p^n}$ should be considered in order to achieve the highest level of security when constructing a pairing?*

The goal is to find the optimal p and n that answers the above question. Note that pairings always come with a given parameter that indicates whether the prime-order subgroup of $\mathcal{E}$ is large. More precisely, this parameter ρ is defined as $\rho = \log p / \log r$ where r is the size of the relevant prime-order subgroup of $\mathcal{E}$ over $\mathbb{F}_p$. In all the known constructions, we have $\rho \in [1, 2]$.

3.7.1 Landing at $p = L_Q(1/3)$ is not as natural as it seems

The fastest known algorithm to solve the DLP on elliptic curves is Pollard rho with a running-time of $O(\sqrt{r})$, which means $O(p^{1/2\rho})$. In order to optimize the security of the scheme that uses such a pairing, a naive and common approach is to balance the two asymptotic complexities, namely $p^{1/2\rho}$

and $L_Q(1/3)$. This would result in $p = L_Q(1/3)$. This equality is not as simple to justify. In the FFS algorithm, the cost of sieving and linear algebra are not taken to be equal, which is a common hypothesis made in the complexity analyses of NFS for example. Assuming this equality would potentially lead to worse complexities. For the same reason, equalizing the cost of the DLPs on the elliptic curve and on the finite field may miss other better options. Interestingly enough, we need the full comprehension of asymptotic complexities at this boundary case to understand why we consider finite fields of this size.

In order to avoid Quasi-Polynomial algorithms, it is clear that one must choose a characteristic $p \geqslant M_Q(1/3, (2/3)^{1/3} \log 2)$. Since FFS and all the variants of NFS have a complexity in $L_Q(1/3, c)$, we then look for finite fields for which the algorithms give the largest c. We distinguish five different areas:

1. **Small characteristic when $p \geqslant M_Q(1/3, (2/3)^{1/3} \log 2)$**. FFS reaches a complexity with $c = (32/9)^{1/3} \approx 1.53$, or lower if n is composite.

2. **Boundary case studied in this thesis**. Various algorithms coexist. When considering the complexity of the optimal algorithms, c roughly varies from 1.16 to 2.46. Note that 2.46 is the lowest complexity reached at the crossover point between FFS and MNFS-$\mathcal{A}$ when nothing is known about p and n.

3. **Medium characteristic**. The lowest complexity in the general case is reached by MNFS-$\mathcal{A}$, giving $c \approx 2.15$.

4. **Boundary case between medium and large characteristics**. The lowest complexity in the general case is reached by MNFS-$\mathcal{A}$ giving here $c \approx 1.70$.

5. **Large characteristic**. The lowest complexities in the general case are reached by MNFS-$\mathcal{A}$ or MTNFS giving here $c \approx 1.90$.

Thus, we see that the best choice is indeed $p = L_Q(1/3)$ so one can expect to reach the highest complexities for DLPs, in particular higher than $L_Q(1/3, 2.15)$, the lowest complexity achieved in medium characteristic.

3.7.2 Fine tuning of c_p to get the highest security

Let us now find c_p that optimizes the security. Let $\mathcal{C}_\mathcal{E}$ (resp. $\mathcal{C}_{\mathbb{F}_Q}$) be the cost of the discrete logarithm computation on the subgroup of the elliptic curve $\mathcal{E}$ (resp. the finite field $\mathbb{F}_Q$). On one hand, we have $\mathcal{C}_\mathcal{E} = p^{1/2\rho}$. This can be rewritten as $\mathcal{C}_\mathcal{E} = L_Q(1/3, c_p/2\rho)$. For ρ fixed, $\mathcal{C}_\mathcal{E}$ is an increasing function of c_p.

On the other hand, the best algorithm to compute discrete logarithms in a finite field depends on three parameters: the size of the characteristic p, the form of p and whether the extension degree n is composite.

General case. Assuming nothing about n and p, the best variant of NFS at this boundary case is MNFS-$\mathcal{A}$. Thus, we have

$$\mathcal{C}_{\mathbb{F}_Q} = \begin{cases} L_Q(1/3, c_{\text{FFS}}(c_p)), & \text{when } c_p \leq \sigma \\ L_Q(1/3, c_{\text{MNFS-}\mathcal{A}}(c_p)), & \text{when } c_p \geq \sigma \end{cases}$$

where $c_{\text{algo}}(c_p)$ is the constant in the L_Q expression of the complexity of the algorithm "algo", and σ is the crossover value of c_p between FFS and MNFS-$\mathcal{A}$.

We then want to find the value of c_p that maximizes $\min(\mathcal{C}_\mathcal{E}, \mathcal{C}_{\mathbb{F}_Q})$. Figure 3.7 shows how the relevant algorithms varies with respect to c_p. Note that the crossover point between $\mathcal{C}_\mathcal{E}$ and $\mathcal{C}_{\mathbb{F}_Q}$ is not with FFS: we just need to compare $\mathcal{C}_\mathcal{E}$ with the complexity of MNFS-$\mathcal{A}$. The latter being a decreasing function with respect to c_p, whereas $\mathcal{C}_\mathcal{E}$ is an increasing function, we conclude that the highest complexities are given at the crossover points between these curves.

For $\rho = 1$, the optimal choice is $p = L_Q(1/3, 4.45)$, which results in an asymptotic complexity in $L_Q(1/3, 2.23)$. For $\rho = 2$, the optimal choice is $p = L_Q(1/3, 8.77)$ resulting in a complexity in $L_Q(1/3, 2.19)$, see Tables 3.5 and 3.6. Increasing ρ from 1 to 2 increases the optimal value of c_p, and thus the asymptotic complexity decreases. This is illustrated in Figure 3.7.
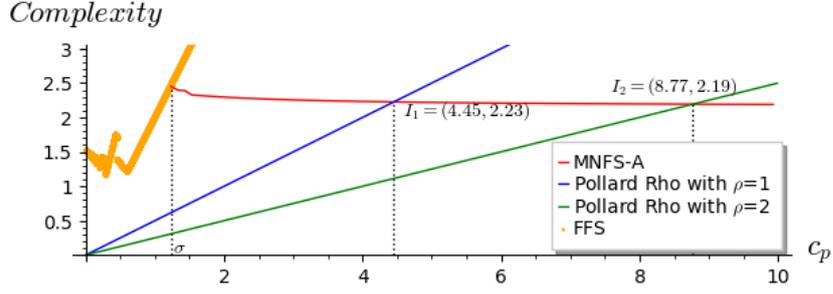
Figure 3.7: Comparing the complexities of FFS, MNFS-$\mathcal{A}$ and Pollard rho for $\rho = 1$ and $\rho = 2$. I_1 and I_2 are the crossover points of $\mathcal{C}_{\mathcal{E}}$ and $\mathcal{C}_{\mathbb{F}_Q}$.
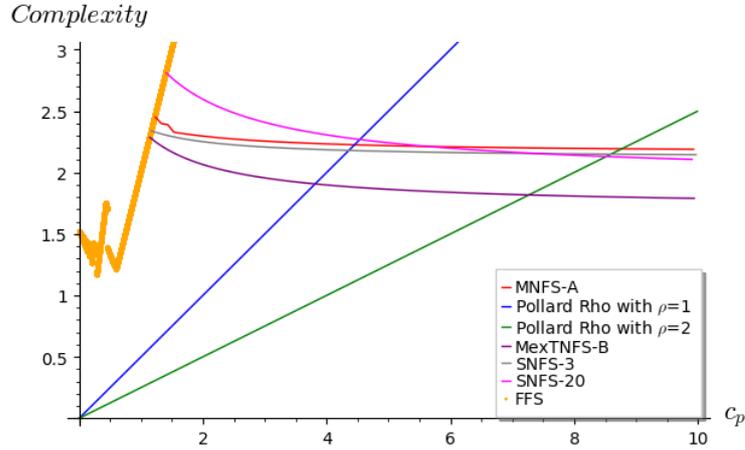
Figure 3.8: Comparing the complexities of FFS, MNFS-$\mathcal{A}$, MexTNFS-B, SNFS-3, SNFS-20 and Pollard rho for $\rho = 1$ and $\rho = 2$. The complexity of MexTNFS-B is always below MNFS-$\mathcal{A}$ which means that a composite n leads to faster algorithms. Yet the complexity of SNFS depends on the value of λ, which is not always below MNFS-$\mathcal{A}$. This means that having a special p does not always decrease the security of the related pairing scheme.

If the extension degree n is composite. The best option as an adversary is to use MexTNFS-$\mathcal{B}$. Its complexity is a decreasing function below the complexity of MNFS-$\mathcal{A}$, see Figure 3.8. Thus, the strategy remains the same. With $\rho = 1$ and $c_p = 3.81$ we obtain an asymptotic complexity in $L_Q(1/3, 1.91)$. With $\rho = 2$ and $c_p = 7.27$ we have a complexity in $L_Q(1/3, 1.82)$, see Tables 3.5 and 3.6. This is illustrated in Figure 3.8.

Special sparse characteristics can be used! When p is given by the evaluation of a polynomial of low degree λ, SNFS is applicable. Yet Figure 3.6 shows that SNFS is not always a faster option than MNFS-$\mathcal{A}$. The behavior of SNFS with regards to MNFS-$\mathcal{A}$ depends on λ:

- If $\lambda = 2$ or $\lambda \geq 29$, then MNFS-$\mathcal{A}$ is faster than the related SNFS for all ρ.

- If $3 \leq \lambda \leq 16$, the related SNFS is faster than MNFS-$\mathcal{A}$ for all ρ.

- If $17 \leq \lambda \leq 28$, the best choice depends on ρ. For instance, if $\lambda = 20$ MNFS-$\mathcal{A}$ is faster if $\rho \leq 1.3$ but SNFS becomes faster if $1.3 \leq \rho$, see Figure 3.9.

Surprisingly enough, this means that we can construct a pairing with a special sparse characteristic without asymptotically decreasing the security of the pairing. For instance, with $\lambda = 20$, $\rho = 1$, the best
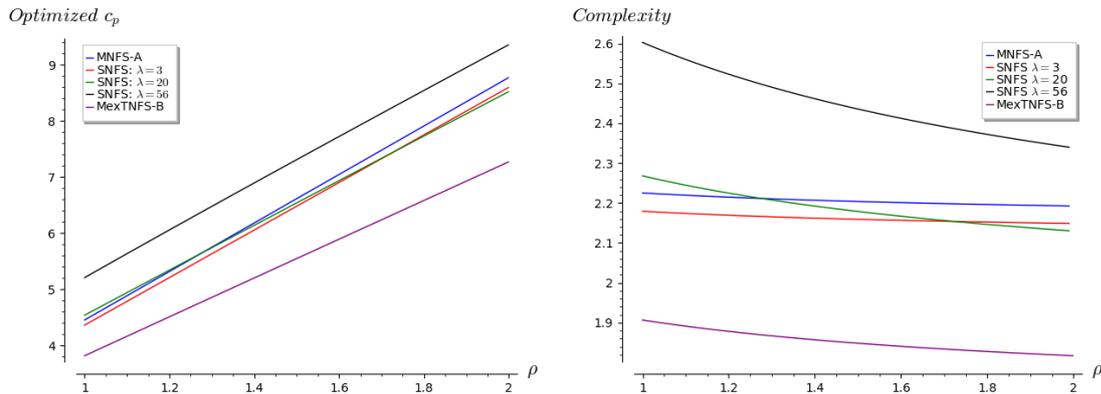
Figure 3.9: Increasing ρ makes the security decrease. The first figure gives optimized values of c_p as a function of ρ, and the second figure shows the second constant in the complexities, as a function of ρ, depending on the algorithm.

	normal p	special p $\lambda = 20$	special p $\lambda = 3$
n prime	$c_p = 4.45$, $c_{\mathbf{MNFS\text{-}\mathcal{A}}} = 2.23$		$c_p = 4.36$, $c_{\mathbf{SNFS\text{-}3}} = 2.18$
n composite	$c_p = 3.81$, $c_{\mathbf{MexTNFS\text{-}B}} = 1.91$		

Table 3.5: Optimal choices for pairing constructions with $\rho = 1$, depending on the form of p and n. Each cell gives the value c_p determining p as $p = L_Q(1/3, c_p)$, together with $c_{\mathbf{algo}}$, which gives the best asymptotic complexity $L_Q(1/3, c_{\mathbf{algo}})$ reached by the algorithm **algo** for the related case.

option is to take $c_p = 4.45$. This gives a complexity in $L_Q(1/3, 2.23)$, which is the one obtained with a normal p of the same size. But for $\lambda = 20$ and $\rho = 2$ the security gets weaker than in the normal case: taking $c_p = 8.51$ allows to decrease the complexity from $L_Q(1/3, 2.19)$ (for a normal p) to $L_Q(1/3, 2.13)$ (for this special p), see Table 3.6.

Combining special p and composite n. We saw in Section 3.5.3 that combining SNFS and exTNFS-$\mathcal{B}$ is not possible at this boundary case. Since (M)exTNFS-$\mathcal{B}$ is always lower than SNFS for the values of c_p considered, with both n composite and p special, the best option is to ignore the form of p, and apply MexTNFS-$\mathcal{B}$.

3.7.3 Conclusion

We studied all possible cases regarding p and n in order to extract the optimized values of c_p, leading to the highest asymptotic security of the related pairing. Tables 3.5 and 3.6 summarize these complexities depending on what is known about p and n. We give our results for $\rho = 1$ and $\rho = 2$. Increasing ρ decreases the complexities and thus the security of pairings, so the values in Table 3.5 are upperbounds on the asymptotic complexities of all currently known pairing constructions. Note however that $\rho = 1$ is achieved with some well-known efficient pairing friendly curves such as MNT or BN curves, yet we emphasize that these families are not asymptotic and, to the best of our knowledge, designing an efficient asymptotic family of pairings reaching $\rho = 1$ is still an open question. The best asymptotic security is given with $\rho = 1$, n prime, and $p = L_Q(1/3, 4.45)$, with p either normal or the evaluation of a degree d polynomial, with $d \geq 29$ or $d = 2$. The asymptotic complexities of all relevant attacks are in $L_Q(1/3, 2.23)$.

	normal p	special p $\lambda = 20$	special p $\lambda = 3$
n prime	$c_p = 8.77$, $c_{\mathbf{MNFS\text{-}}\mathcal{A}} = 2.19$	$c_p = 8.51$, $c_{\mathbf{SNFS\text{-}20}} = 2.13$	$c_p = 8.59$, $c_{\mathbf{SNFS\text{-}3}} = 2.15$
n composite	$c_p = 7.27$, $c_{\mathbf{MexTNFS\text{-}B}} = 1.82$		

Table 3.6: Optimal choices for pairing constructions with $\rho = 2$, depending on the form of p and n. Each cell gives the value c_p determining p as $p = L_Q(1/3, c_p)$, together with $c_{\mathbf{algo}}$, which gives the best asymptotic complexity $L_Q(1/3, c_{\mathbf{algo}})$ reached by the algorithm **algo** for the related case.

Chapter 4

Enumeration algorithms for algebraic sieving in TNFS

In this chapter, we investigate the relation collection step of the Tower Number Field Sieve (TNFS) algorithm. Because sieving in TNFS requires a higher dimensional space than in the classical NFS, we propose a new efficient sieving algorithm for higher dimensions based on Schnorr-Euchner's lattice enumeration algorithm. On the contrary to previous higher-dimension sieving algorithms, we also use a d-dimensional sphere instead of a d-dimensional orthotope as search space for relations. This allows us to efficiently sieve in dimensions as large as 6 for example. We anchor our sieving algorithm in the general context of relation collection explaining how it can be combined with other algorithms such as ECM and batch smoothness to provide an optimized relation collection step. Finally, we incorporate technical details such as Schirokauer maps, virtual logarithms and duplicate relations in order to present a complete overview of the TNFS algorithm. This chapter serves as basis to the following Chapter 5 where we describe the first implementation of the TNFS algorithm and a 521-bit record computation.

Contents

4.1 Introduction

4.1.1 Motivation

The Number Field Sieve algorithm and its variants are the fastest known algorithms to solve the discrete logarithm problem in finite fields of medium and large characteristics. One of these variants, the Tower Number Field Sieve (TNFS), is a generalization of NFS that exploits towers of number fields, hence the name. The main difference with NFS comes from the representation of the target field $\mathbb{F}_{p^n}$. Whereas in the classical NFS setup, the finite field $\mathbb{F}_{p^n}$ is represented as the quotient field $\mathbb{F}_p[x]/(f)$ where f is a polynomial of degree n over $\mathbb{F}_p$, in the TNFS setup, we have $\mathbb{F}_{p^n} \cong \mathcal{R}/p\mathcal{R}$ where $\mathcal{R}$ is the ring defined as the quotient $\mathbb{Z}[t]/h(t)$, and h is also a degree n irreducible polynomial over $\mathbb{F}_p$.

Originally proposed by Schirokauer [Sch00], the TNFS algorithm was reinvestigated in [BGK15] where the authors simplified Schirokauer's original presentation of the algorithm and incorporated more recent tools such as Schirokauer maps and virtual logarithms. They also showed that the asymptotic complexity of TNFS in large characteristic is $L_{p^n}(1/3, \sqrt[3]{64/9})$, similarly as NFS. In small and medium characteristics, the complexity of TNFS is greater than the usual $L_{p^n}(1/3)$ and thus this algorithm is only considered beyond the medium case.

The Tower NFS algorithm was then expanded to the extended TNFS in [KB16, KJ17] where the extension degree $n = \eta\kappa$ is composite. This allows the extended variant to have an $L_{p^n}(1/3)$ complexity also in medium characteristic. In this case, the overall complexity of exTNFS is as low as $L_{p^n}(1/3, \sqrt[3]{48/9})$ and in large characteristic it equals $L_{p^n}(1/3, \sqrt[3]{64/9})$. These complexities are summarized in Table 4.1. Both TNFS and exTNFS can be coupled with a multiple field variant and a special variant but we do not address these variants in this chapter.

Algorithm	medium char	2nd boundary	Large char
NFS	96	48	64
TNFS	–	–	64
exTNFS	48	48	64

Table 4.1: The best complexities of each algorithm in medium and large characteristics. Since the complexity is expressed as $L_{p^n}(1/3, \sqrt[3]{c/9})$ we only report the value c.

One can see from these complexities that for NFS, the medium characteristic is at least as hard as the large characteristic. This remains true for most variants.

However, a noticeable exception to this observation lies in the exTNFS algorithm. Indeed, in medium characteristic the algorithm has better complexity than in large characteristic which makes it a promising candidate for computational records in finite fields of medium characteristic. So far, excluding the small characteristic, all computational records were performed using the Number Field Sieve algorithm with occasionally a special variant or FFS. The Tower Number Field Sieve has never been implemented (up to this thesis) despite promising perspectives. Such an implementation would also provide practical insight on security parameters for pairing-based protocols where the extension degree of the finite field is often composite. In the rest of the chapter, we assume the degree n is composite, i.e., $n = \eta\kappa$.

A major obstacle to an efficient implementation of the (extended) Tower Number Field Sieve algorithm is the relation collection step. Indeed, whereas NFS requires sieving through $(a, b) \in \mathbb{Z}^2$ pairs, the tower setup requires sieving through $(a(\iota), b(\iota))$-pairs, i.e., degree $\eta - 1$ polynomials with bounded coefficients. This implies that sieving must occur in a space of dimension greater than 2 for which an efficient algorithm must be found. We recall that in dimension 2, Franke and Kleinjung [FK05] proposed an efficient algorithm used in all previous records.

For higher dimensions, the transition vectors method from Grémy [Gré18] and a recursive plane method proposed by McGuire and Robinson [MR21] have been tested in dimension 3 and used for record computations using NFS. However, the efficiency of their algorithms for even higher dimensions is questionable.

The focus of this chapter is thus to introduce an efficient sieving algorithm in higher dimensions which will allow us to implement TNFS and perform the first record computation with this variant.

4.1.2 Contribution

More specifically, we propose the following contributions.

Sieving in a high dimensional sphere instead of an orthotope. All sieving algorithms so far considered a product of intervals as search space $\mathcal{S}$. Indeed, whether a candidate relation is characterized by an (a, b)-pair or an $(a(\iota), b(\iota))$-pair with more than two coefficients, every coefficient is bounded separately in an interval $[-H_1^i, H_2^i]$ for $i = 1, 2, \cdots, d$ where d is the total number of coefficients. Hence, the search space considered is of the form $\mathcal{S} = [-H_1^1, H_2^1] \times \cdots \times [-H_1^d, H_2^d]$.

In this chapter, we argue that when $d \geq 3$, the shape of the search space $\mathcal{S}$ must be adequately chosen in order to have an efficient sieving algorithm. More precisely, we consider a d-sphere instead of a d-orthotope and argue why we believe this choice leads to a more efficient algorithm when the dimension grows.

Adapting a lattice enumeration algorithm to the context of TNFS. In order to fully exploit the new search space, we adapt a known lattice algorithm to the context of TNFS: Schnorr-Euchner's enumeration algorithm [SE94]. The latter enumerates all the vectors of an input lattice $\mathcal{L}$ within a d-dimensional sphere S_d. We modify this algorithm in order to answer the requirements of the sieving step and provide a complete pseudo-code of our modified algorithm. This algorithm remains competitive when the dimension grows and also provides an exhaustive search of all the vectors in $\mathcal{L} \cap S_d$. Finally we compare this new sieving method with other previously mentioned algorithms.

A complexity analysis of the relation collection step in TNFS. Finally, we anchor this sieving algorithm in the context of the entire relation collection step. Sieving algorithms are usually combined in practice with batch algorithms and ECM to provide the most efficient relation collection. We explain the details of the relation collection step, including Schirokauer maps and duplicate relations, as a preparation for the next Chapter 5 which will give details of the actual implementation and the record computation.

4.2 The Tower Number Field Sieve

4.2.1 Mathematical setup

The classical tower of number fields that illustrates the TNFS setup considers the intermediate number field $\mathbb{Q}(\iota)$ where ι is a root of h, an irreducible polynomial mod p. Above this number field are set the two number fields $K_1 = \mathbb{Q}(\iota)[x]/f_1(x)$ and $K_2 = \mathbb{Q}(\iota)[x]/f_2(x)$ where f_1, f_2 are irreducible polynomials over $\mathcal{R} = \mathbb{Z}[\iota]$ that share an irreducible factor modulo the unique ideal $\mathfrak{p}$ over p in $\mathbb{Q}(\iota)$. We will write $\mathcal{O}_i$ their ring of integers for $i = 1, 2$. Let α_1 and α_2 be roots of f_1 and f_2 in K_1 and K_2. This construction is illustrated in Figure 4.1.

$$
\begin{array}{cc}
K_1 & K_2 \\
f_1 \diagdown & \diagup f_2 \\
\mathcal{R} \subset \mathbb{Q}(\iota) & \\
h \;\Big| & \\
\mathbb{Q} &
\end{array}
$$
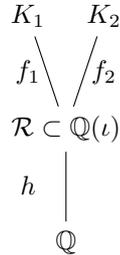
Figure 4.1: Tower of Number Fields.

Because of the conditions on the polynomials h, f_1 and f_2, there exists two ring homomorphisms from $\mathcal{R}[x] = \mathbb{Z}[\iota][x]$ to the target finite field $\mathbb{F}_{p^n}$ through the number fields K_1 and K_2. This allows to build a commutative diagram as explained in Chapter 1 and shown in Figure 4.2.
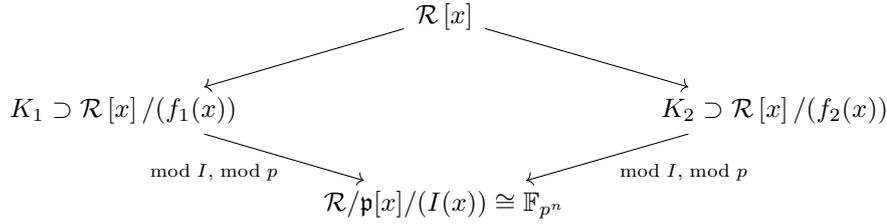
Figure 4.2: Commutative diagram of TNFS.

> **Question 41.** *What are the advantages of (ex)TNFS?*

When the degree n is composite, *i.e.*, $n = \eta\kappa$, the target finite field $\mathbb{F}_{p^n} = \mathbb{F}_{p^{\eta\kappa}}$ can be viewed as $\mathbb{F}_{P^\kappa}$ where P is a prime of the same bitsize as p^η. Thus, the complexity of a computation done in medium characteristic with exTNFS can be viewed similarly as the complexity of a computation done with NFS at the boundary case between medium and large characteristic, meaning with a smaller c constant in the L_{p^n}-notation, see Table 4.1.

4.2.2 A step by step walk through TNFS

The TNFS algorithm then follows similar steps as any index calculus algorithm. The main differences with the Number Field Sieve take place in the polynomial selection step and the collection of relations.

Polynomial selection. On the contrary of NFS which uses only two polynomials f_1 and f_2 to define the number fields, three polynomials must be selected for this algorithm, namely h, f_1 and f_2. The polynomial h must be of degree η and irreducible modulo p to ensure the uniqueness of the ideal $\mathfrak{p}$ over p in $\mathcal{R}$. Ideally one would choose a unitary h with small coefficients and such that the inverse of the Dedekind zeta function (implemented in Sage for example) is close to 1. Indeed, as we will see in Section 4.3.5, similarly as for NFS, non-coprime ideals produce equivalent relations which are useless for the linear algebra step. Moreover, the proportion of coprime ideals is given by

$$\prod_{\mathfrak{q} \text{ prime ideal in } \mathbb{Q}(\iota)} \left(1 - \frac{1}{N(\mathfrak{q})^2}\right) = \frac{1}{\zeta_{\mathbb{Q}(\iota)}(2)},$$

where $\zeta_{\mathbb{Q}(\iota)}$ is the Dedekind zeta function. Hence, in order to reduce the amount of duplicate relations produced because of these non-coprime ideals, one can choose an h with $\zeta_{\mathbb{Q}(\iota)}(2)$ as close to 1 as possible.

The polynomials f_1 and f_2 are selected using classic polynomial selections such as the Conjugation method, JLSV or the Sarkar-Singh method.

> **Question 42.** *Which polynomial selection method should one use to select f_1 and f_2?*

We assume here we do not consider sparse characteristics. Recall that JLSV produces polynomials f_1 and f_2 of same degree, say d, and with coefficients of size $O(\sqrt{p})$ for both polynomials.

On the other hand, the Conjugation method has more unbalanced parameters. The polynomial f_1 has a larger degree $2d$ and small coefficient in $O(1)$ and f_2 has degree d (the same as for JLSV) and coefficients of size $O(\sqrt{p})$.

Thus in order to decide which method to use, one should compare the size of the norms $N_1(\phi)$ and $N_2(\phi)$ for elements $\phi \in \mathcal{R}[x]$. If $N_1(\phi) < N_2(\phi)$, then one should choose the Conjugation method. However, if $N_2(\phi) < N_1(\phi)$, then one should choose the JLSV method since the polynomial f_2, which shares the same properties as polynomials outputted from JLSV, has the smaller norm.

In [SS19], Sarkar and Singh propose a polynomial selection $\mathcal{D}$ for exTNFS (already mentioned in Chapter 3) which generalizes previous methods such as Conjugation and GJL. They show that using this polynomial selection improves the overall complexity of exTNFS for certain ranges of finite fields in medium characteristics.

The polynomials we will use in Chapter 5 for our 521-bit computation come from the Conjugation method. Further details on how they were chosen will be given in Section 5.2.

Remark 8. *In general, the coefficients of the polynomials f_i are taken in $\mathcal{R}$. However, as was noted in [BGK15], no significant gain is observed by doing so and thus we will consider the case $f_i \in \mathbb{Z}[x]$ in the rest of the chapter.*

In NFS, the quality of the polynomials can be refined with a quantity known as the Murphy-α value. We refer to [GS21] for details about this value adapted to the TNFS context.

Relation collection. Recall that the goal of the relation collection step is to select among the set of linear polynomials $\phi(x, \iota) = a(\iota) - b(\iota)x \in \mathcal{R}[x]$ at the top of the diagram the candidates which will produce a relation.

As explained in Chapter 1, a relation is found if the polynomial $\phi(x, \iota)$ mapped to K_1 and K_2 factors into products of ideals of small norms in both number fields. The ideals of small norms that occur in these factorizations constitute the factor basis. More precisely, the factor basis is defined as $\mathcal{F} = \mathcal{F}_1 \cup \mathcal{F}_2$ with

$$\mathcal{F}_i(B) = \{\text{prime ideals of } \mathcal{O}_i \text{ of norm} \leq B, \text{ whose inertia degree over } \mathbb{Q}(\iota) \text{ is one}\},$$

for $i = 1, 2$.

Remark 9. *The prime ideals in the factorization of $(a(\iota) - b(\iota)\alpha_i)O_i$ can have degree at most $\deg \phi$. Thus the ideals in the factor basis have degree at most 1 in the variable x. However, in the specific context of TNFS, the polynomial ϕ is a polynomial in two variables: x and ι. Thus, these ideals can have degree in ι up to degree of h.*

Question 43. *Why do we only consider ideals of degree 1 in ι?*

In theory, we only lose a constant factor in the smoothness probabilities when excluding the ideals of degree greater than 1. In practice, if we encounter a prime ideal that has a degree greater than 1 in the variable ι, we keep the relation, as will be explained in Chapter 5. Otherwise, we would throw away too many relations. The same holds for ideals dividing the index ideal $[\mathcal{O}_i : \mathcal{O}_i[\alpha_i]]$ but there are only finitely many and they can be handled separately.

The representation of these ideals of degree 1 in the context of TNFS is summarized in the following result.

Proposition 1 (Proposition 1 [BGK15])**.** *Let $\mathbb{Q}(\iota)$ be a number field and let $\mathcal{O}_\iota$ be its ring of integers. Let f be a monic irreducible polynomial in $\mathbb{Q}(\iota)[x]$, and denote by α one of its roots. We denote by $K = \mathbb{Q}(\alpha, \iota)$ the corresponding extension field, and $\mathcal{O}_f$ its ring of integers.*
If $\mathfrak{q}$ is a prime ideal of $\mathcal{O}_\iota$ not dividing the index-ideal $[\mathcal{O}_f : \mathcal{O}_\iota[\alpha]]$, then the following statement holds.

1. *The prime ideals of $\mathcal{O}_f$ above $\mathfrak{q}$ are all the ideals of the form*

$$\mathfrak{Q} = (\mathfrak{q}, T(\alpha)),$$

   where $T(x)$ are the lifts to $\mathcal{O}_\iota[x]$ of the irreducible factors of f in $\mathcal{O}_\iota/\mathfrak{q}[x]$. Moreover, $\deg \mathfrak{Q} = \deg T$.

2. *If $a(t), b(t) \in \mathbb{Z}[t]$ are such that $\mathfrak{q}$ divides $N_{K/\mathbb{Q}(\iota)}(a(\iota) - b(\iota)\alpha)$ and $a(\iota)\mathcal{O}_\iota + b(\iota)\mathcal{O}_\iota = \mathcal{O}_\iota$, then the unique ideal of $\mathcal{O}_f$ above $\mathfrak{q}$ which divides $a(\iota) - b(\iota)\alpha$ is $\mathfrak{Q} = (\mathfrak{q}, \alpha - r(\iota))$ with $r \equiv a(\iota)/b(\iota)$ (mod $\mathfrak{q}$).*

To verify the B-smoothness of these ideals, one computes as usual the absolute norms $N_1(a(\iota) - b(\iota)\alpha_1)$ and $N_2(a(\iota) - b(\iota)\alpha_2)$ using their definitions with the notion of resultant

$$N_1(a(\iota) - b(\iota)\alpha_1) = \text{Res}_t(\text{Res}_x(a(t) - b(t)x, f_1(x)), h(t))$$

and

$$N_2(a(\iota) - b(\iota)\alpha_2) = \mathrm{Res}_t(\mathrm{Res}_x(a(t) - b(t)x, f_2(x)), h(t)).$$

These formulas must be adapted when the polynomials f_i are not monic by including the leading coefficient raised to the degree of the polynomial, as seen in Chapter 1.

This allows to verify the B-smoothness over integer values. As usual, one collects enough relations before constructing a linear system of equations. The unknowns of these equations are the virtual logarithms of the ideals of the factor basis. The notion of virtual logarithms and a more explicit construction of the system is given in Section 4.2.3.

Linear algebra: Wiedemann's algorithm [Wie86]. The goal of Wiedemann's algorithm is to compute kernel vectors of a matrix over a finite field. Let M be the $N \times N$ input matrix, in our context the matrix of relations after filtering. The output of the algorithm is a kernel vector $\mathbf{w} \neq \mathbf{0}$, *i.e.*, a vector $\mathbf{w}$ such that $M\mathbf{w} = \mathbf{0}$. We expect this vector to exist because M is constructed so that the virtual logarithms of the ideals give such a solution.

The algorithm relies on the use of the minimal polynomial of the matrix M. We recall that the minimal polynomial of M, denoted μ_M, is the polynomial of smallest degree that annihilates M, meaning $\mu_M(M) = 0$. We will write $\mu_M(x) = \sum_{i=0}^{d} c_i x^i$ where the degree d of μ_M is less than N by Cayley-Hamilton's theorem. First one can note that the constant coefficient c_0 is necessarily equal to zero. Indeed, we know that $\sum_{i=0}^{d} c_i M^i \mathbf{w} = 0$ and thus $c_0 \mathbf{w} = -\sum_{i=1}^{d} c_i M^i \mathbf{w} = 0$. Hence $c_0 = 0$.

Let ℓ be a large prime, in our context the prime ℓ is the largest factor of $p^n - 1$. Indeed, discrete logarithm computations are done modulo ℓ, see Pohlig-Hellman's reduction described in Chapter 1. Because $\mu_M(M) = 0$, we have

$$M\left(\sum_{i=1}^{d} c_i M^{i-1}\right)\mathbf{z} = 0,$$

for any vector $\mathbf{z}$ randomly chosen with coefficients in $\mathbb{Z}/\ell\mathbb{Z}$. Hence one solution would be to consider $\mathbf{w} = \left(\sum_{i=1}^{d} c_i M^{i-1}\right)\mathbf{z}$. Of course, the vector $\mathbf{w}$ must be non-zero and this is the case if $\mathbf{z}$ is not in the kernel of the matrix $\left(\sum_{i=1}^{d} c_i M^{i-1}\right)$. In order to fully determine the solution $\mathbf{w}$, it remains to determine the sequence $\{c_i\}_{i=1}^{d}$ and find a suitable vector $\mathbf{z}$.

Question 44. *How do we find the coefficients c_i?*

Instead of directly computing μ_M, which is costly, Wiedemann's algorithm considers the sequence $\lambda_i = \mathbf{a}^\intercal M^i \mathbf{b}$ for $i = 1, 2, \cdots, 2N$ where $\mathbf{a}, \mathbf{b}$ are two random vectors of size N with coefficients in $\mathbb{Z}/\ell\mathbb{Z}$. This is known as the Krylov sequence. Because $\sum_{i=0}^{N} c_i M^i = 0$ (from Cayley-Hamilton's theorem), where the c_i are the coefficients of the minimal polynomial and one can add $c_i = 0$ for $i > d$, we also have

$$\sum_{i=1}^{N} c_i \mathbf{a}^\intercal M^{i+j} \mathbf{b} = 0,$$

for any $j > 0$. A key element in the algorithm is the fact that the sequence $\{\mathbf{a}^\intercal M^i \mathbf{b}\}_{i=0}^{2N}$ previously computed is generated by a linear recursion. The Berlekamp-Massey [Ber15, Mas69] algorithm then precisely allows to find the minimal polynomial of this sequence. Hence, we can find the coefficients c_i such that

$$\sum_{i=1}^{N} c_i \mathbf{a}^\intercal M^{i+j} \mathbf{b} = 0.$$

Question 45. *Does this implies that the c_i satisfy $\sum_{i=0}^{N} c_i M^i = 0$?*

Wiedemann's method relies on the assumption that the minimal polynomial of the sequence $\{\mathbf{a}^\intercal M^i \mathbf{b}\}_{i=0}^{2N}$ is equal to the minimal polynomial of M with high probability for random vectors $\mathbf{a}$ and $\mathbf{b}$. A result from Kaltofen [Kal95][Theorem 5] proves that this is indeed the case. Hence, the d coefficients c_i necessary to find $\mathbf{w}$ can be extracted from the $2N$ coefficients $\{\mathbf{a}^\intercal M^i \mathbf{b}\}_{i=1}^{2N}$.

Question 46. *How do we reconstruct the final solution?*

Recall that we are looking for a vector $\mathbf{w} \neq \mathbf{0}$ such that $M\mathbf{w} = 0$. Now that we have the coefficients c_i for $i = 1, 2, \cdots, d$, reconstructing $\mathbf{w}$ is done as explained in the beginning. For a random vector $\mathbf{z}$ with coefficients in $\mathbb{Z}/\ell\mathbb{Z}$, we have $\mathbf{w} = \left(\sum_{i=1}^{d} c_i M^{i-1} \right) \mathbf{z}$.

Question 47. *What is the complexity of Wiedemann's algorithm?*

The main cost of Wiedemann's algorithm comes from the cost of multiplying a sparse matrix by a vector. This operations has a cost of $O(\lambda N)$ where λ is the average number of non-zero coefficients per row. Hence, computing the sequence $\{\mathbf{a}^{\mathsf{T}} M^i \mathbf{b}\}_{i=1}^{2N}$ has a cost in $O(\lambda N^2)$ and so does computing the final solution. Since Berlekamp-Massey algorithm has variants with a quasi-linear complexity, the overall cost of Wiedemann's algorithm is in $O(\lambda N^2)$, which is the cost of linear algebra already mentioned in Chapter 3.

A major improvement to Wiedemann's algorithm is the use of a block variant which allows to parallelize part of the algorithm. This is a significant advantage for record computations. This block variant will be described in Chapter 5 as we use it for our record computation.

Descent in TNFS.   Recall that the final step of TNFS consists in finding the discrete logarithm of the target element. This step is subdivided into two substeps: a smoothing step and a descent step. The smoothing step is an iterative process where the target element t is randomized by considering $s = g^x t \in \mathbb{F}_{p^n}^*$ for an exponent x chosen uniformly at random. Values for x are tested until s lifted back to one of the number fields K_i is B_i-smooth for a smoothness bound $B_i > B$.

The second step consists in decomposing every factor of the lifted value of s, in our case prime ideals with norms less than B_i (but usually greater than B) into elements of the factor basis for which we now know the virtual logarithms. This process creates descent trees where the root is an ideal coming from the smoothing step, and the nodes are ideals that get smaller and smaller as they go deeper. The leaves are ultimately elements of the factor basis. The edges of the tree are defined as follows: for every node, there exists an equation between the ideal of the node and all the ideals of its children.

We focus here on an improvement given by Guillevic in [Gui19] for the first of these two steps, *i.e.*, the smoothing step. The goal is to improve the B_i-smoothness probability of the lift of $s \in \mathbb{F}_{p^n}^*$ to K_i. Let s' denote this lift. Note that this lift to K_i is done on a unique side, usually the one with smaller norms.

Let φ_i be the map from K_i to $\mathbb{F}_{p^n}$. Instead of requiring that $\varphi_i(s') = s$, the algorithm looks for $s' \in K_i$ such that $\varphi_i(s') = su$ for $u \in \mathbb{F}_{p^n}$. Indeed, since $\log(u) = 0$ we will still have $\log(\varphi_i(s')) = \log(s)$.

Now, the set of pre-images such that $\varphi(s') = su$ forms a lattice. The general idea of this improvement is then to construct the lattice spanned by these pre-images and reduce its basis to obtain a short vector. The shortest vector of the reduced basis will define an element of K_i with potentially small norm which is precisely the s' we are looking for.

Question 48. *How is the lattice basis constructed?*

We look at the $\mathbb{F}_p$-vector space $\{su\}$ for $u \in \mathbb{F}_{p^n}$ of dimension η with basis $\mathcal{B} = < s, s\iota, s\iota^2, \cdots, s\iota^{\eta-1} >$ where ι is a root of the polynomial h that defined $\mathbb{F}_{p^n} = \mathbb{F}_p[\iota]/(h)$. Using Gauss reduction's algorithm, we obtain a row-echelon basis of this vector space where each line denotes an s_i candidate.

The lattice basis generates the set of elements of K_i such that $\varphi(s') = 0$ or $\varphi(s') = s_i$. These correspond to either multiples of p (for which the projection will give 0), lifts of the s_i given above, namely s_i' or finally multiples of the polynomial I that defined the target finite field, *i.e.*, $\mathbb{F}_{p^n} = \mathbb{F}_p[x]/(I)$. The following lattice basis can thus be constructed as follows. We give an example for $n = 6$ and $\eta = 3$ to simplify the presentation of the basis where each line corresponds to the coefficients of the polynomials given on the left. We refer to [Gui19][Algorithm 5] for the full algorithm.

	1	ι	ι^2	x	$x\iota$	$x\iota^2$	x^2	$x^2\iota$	$x^2\iota^2$	x^3	$x^3\iota$	$x^3\iota^2$
p	p											
$p\iota$		p										
$p\iota^2$			p									
s'_1	$*$	$*$	$*$	1								
s'_2	$*$	$*$	$*$	$*$	1							
s'_3	$*$	$*$	$*$	$*$	$*$	1						
I	$*$	$*$	$*$	$*$	$*$	$*$	1					
ιI	$*$	$*$	$*$	$*$	$*$	$*$	$*$	1				
$\iota^2 I$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	1			
xI	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	1		
$x\iota I$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	1	
$x\iota^2 I$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	$*$	1

4.2.3 Virtual logarithms and Schirokauer maps

We have already mentioned in Chapter 1 that the relations come from polynomials ϕ at the top of the commutative diagram which give linear equations between the *virtual logarithms of ideals of small norms*. We now explain what these virtual logarithms are and how they are connected to the notion of Schirokauer maps. We describe these concepts in the classical NFS setup to facilitate the presentation.

Recall from Chapter 1 that Pohlig-Hellman's reduction allows to find discrete logarithms in prime order subgroups and then reconstitute the target discrete logarithm in the general group using the Chinese Remainder Theorem. From now on, let ℓ be a prime divisor of $p^n - 1$ and we will consider the computation of logarithms modulo ℓ. Moreover, in the rest of the section, an element $x \in \mathcal{O}_i$ of an intermediate number field K_i mapped to the target field $\mathbb{F}_{p^n}$ using the maps φ_i, see Chapter 1, Figure 1.4, will be denoted $\bar{x}$.

> **Question 49.** *Why do we need virtual logarithms and what are they?*

Let us start with the classical NFS diagram. At the top of the diagram, we begin with a pair (a, b) and we map the quantity $a - bx \in \mathbb{Z}[x]$ into two intermediate number fields K_i. We get $a - bx \pmod{(f_i)} = a - b\alpha_i \in \mathbb{Z}[x]/(f_i) \subset \mathcal{O}_i$ for $i = 1, 2$. Let us assume that the pair (a, b) is B-smooth on both sides. Since the diagram is commutative, we have an equality in the bottom field $\mathbb{F}_{p^n}$ which then gives a relation.

In the intermediate number fields, the ideals $(a - b\alpha_i)$ can be decomposed into a product of first degree prime ideals, *i.e.,* we have

$$(a - b\alpha_1)\mathcal{O}_1 = \prod_j \mathfrak{p_j}^{e_j} \quad \text{and} \quad (a - b\alpha_2)\mathcal{O}_2 = \prod_j \mathfrak{q_j}^{e'_j},$$

where $e_j, e'_j > 0$ are integer values. Because we have assumed B-smoothness on both sides, this means all the ideals $\mathfrak{p}_j, \mathfrak{q}_j$ in the above equations are contained in the factor basis. The multiplicative relations we have at the end of the sieving step are thus of the form

$$(a - b\alpha_1)\mathcal{O}_1 = \prod_j \mathfrak{p_j}^{e_j} \text{ `` } = \text{ '' } \prod_j \mathfrak{q_j}^{e'_j} = (a - b\alpha_2)\mathcal{O}_2$$

where the quantities need first to be mapped to $\mathbb{F}_{p^n}$. For the linear algebra step, we want to transform these multiplicative relations into additive relations of the form

$$e_1 \text{``}\log(\mathfrak{p}_1)\text{''} + \cdots + e_m \text{``}\log(\mathfrak{p}_m)\text{''} = e'_1 \text{``}\log(\mathfrak{q}_1)\text{''} + \cdots + e'_n \text{``}\log(\mathfrak{q}_n)\text{''} \pmod{\ell}.$$

Since we do not know how to compute the quantities "$\log(\mathfrak{p}_j)$" and "$\log(\mathfrak{q}_j)$", we look for an alternative definition of logarithm which still allows us to write an additive relation. This alternative definition is known as a virtual logarithm. We now proceed to explaining the context in which they arise from.

The ideals in the factor basis are not necessarily principal. Let h_i be the (finite) class number of the intermediate number field $\mathbb{Q}[x]/(f_i(x))$ for $i = 1, 2$. Then, we know that $\mathfrak{p_j}^{h_1}$ and $\mathfrak{q_j}^{h_2}$ are principal ideals for all j and thus we can write

$$\mathfrak{p_j}^{h_1} = (\gamma_{\mathfrak{p}_j})\mathcal{O}_1 \quad \text{and} \quad \mathfrak{q_j}^{h_2} = (\gamma_{\mathfrak{q}_j})\mathcal{O}_2.$$

Remark 10. *An important observation is that class numbers h_1, h_2 are computationally hard to obtain and are only used in this context to describe the intuition behind the definition of a virtual logarithm.*

Continuing with the expression of principal ideals, we have

$$(a - b\alpha_1)^{h_1}\mathcal{O}_1 = \prod_j (\gamma_{\mathfrak{p}_j})^{e_j}\mathcal{O}_1 \quad \text{and} \quad (a - b\alpha_2)^{h_2}\mathcal{O}_2 = \prod_j (\gamma_{\mathfrak{q}_j})^{e'_j}\mathcal{O}_2.$$

Since two generators of a same principal ideal are equal up to a unit of K_i, there exist units $u_i \in \mathcal{O}_i$ such that $(a - b\alpha_1)^{h_1} = u_1 \prod(\gamma_{\mathfrak{p}_j}^{e_j})$ and $(a - b\alpha_2)^{h_2} = u_2 \prod(\gamma_{\mathfrak{q}_j}^{e'_j})$. Mapping these elements to $\mathbb{F}_{p^n}$ and taking the log would yield

$$h_1 \log(\overline{a - b\alpha_1}) = \log(\overline{u_1}) + \sum_j e_j \log(\overline{\gamma_{\mathfrak{p}_j}}) \pmod{\ell}$$

and

$$h_2 \log(\overline{a - b\alpha_2}) = \log(\overline{u_2}) + \sum_j e'_j \log(\overline{\gamma_{\mathfrak{q}_j}}) \pmod{\ell}.$$

The quantities in the logarithms on the right-hand-side of the equations are undefined. Let us focus on the units first.

Question 50. *How can we define the quantities $\log(\overline{u_1})$ and $\log(\overline{u_2})$?*

The answer to this question comes with the definition of a virtual logarithm. From Dirichlet's theorem, we know that the group of units $\mathcal{O}_i^*$ of K_i is finitely generated and $\mathcal{O}_i^* \cong \mathbb{Z}/t_i\mathbb{Z} \times \mathbb{Z}^{r_i}$ where the first part corresponds to the torsion and $r_i = n_{i_1} + n_{i_2} - 1$ is the unit rank of K_i with n_{i_1} real roots and $2n_{i_2}$ complex roots for the polynomial f_i.

Assuming that determining a system of fundamental units $\{u_{i_j}\}_{j=1}^{r_i}$ of $\mathcal{O}_i$, the torsion-free part of $\mathcal{O}_i^*$ is easy (which is not!), let κ_i be a root of unity and g_{i_j} and x_i a set of integer exponents such that

$$u_i = \kappa_i^{x_i} \prod_{j=1}^{r_i} u_{i_j}^{g_{i_j}},$$

for $i = 1, 2$. Then, by taking the logarithm of this expression we would have $\log(\overline{u_i}) = x_i \log(\overline{\kappa_i}) + \sum_{j=1}^{r_i} g_{i_j} \log(\overline{u_{i_j}}) \pmod{\ell}$, for $i = 1, 2$. Note that $\log(\overline{\kappa_i}) = 0 \pmod{\ell}$ if ℓ and t_i are coprime. Hence, by substituting the expression of u_i in our previous equations, we now have

$$h_1 \log(\overline{a - b\alpha_1}) = \sum_{j=1}^{r_1} g_{1_j} \log(\overline{u_{1_j}}) + \sum_{j=1}^{m} e_j \log(\overline{\gamma_{\mathfrak{p}_j}}) \pmod{\ell}$$

and

$$h_2 \log(\overline{a - b\alpha_2}) = \sum_{j=1}^{r_2} g_{2_j} \log(\overline{u_{2_j}}) + \sum_{j=1}^{n} e'_j \log(\overline{\gamma_{\mathfrak{q}_j}}) \pmod{\ell}.$$

A relation thus looks like

$$\sum_{j=1}^{r_1} g_{1_j} h_1^{-1} \log(\overline{u_{1_j}}) + \sum_{j=1}^{m} e_j h_1^{-1} \log(\overline{\gamma_{\mathfrak{p}_j}}) - \sum_{j=1}^{r_2} g_{2_j} h_2^{-1} \log(\overline{u_{2_j}}) - \sum_{j=1}^{n} e'_j h_2^{-1} \log(\overline{\gamma_{\mathfrak{q}_j}}) = 0 \pmod{\ell} \quad (4.1)$$

We introduce the following definition.

Definition 21 (Virtual logarithm). *Consider the intermediate number fields K_i with class number h_i for $i = 1, 2$. Let $\mathfrak{p}$ be an ideal in the factor basis $\mathcal{F}$ and let $\mathfrak{p}^{h_i} = (\gamma_{\mathfrak{p}})$. The virtual logarithm of the ideal $\mathfrak{p}$ is given by*

$$\mathrm{vlog}(\mathfrak{p}) = h_i^{-1} \log(\overline{\gamma_{\mathfrak{p}}}) \pmod{\ell}.$$

Similarly, we define the virtual logarithm of fundamental units as

$$\mathrm{vlog}(u_{i_j}) = h_i^{-1} \log(\overline{u_{i_j}}) \pmod{\ell}.$$

Going back to what a relation looks like, Equation 4.1 becomes

$$\sum_{j=1}^{r_1} g_{1_j} \mathrm{vlog}(u_{1_j}) + \sum_{j=1}^{m} e_j \mathrm{vlog}(\mathfrak{p}_j) - \sum_{j=1}^{r_2} g_{2_j} \mathrm{vlog}(u_{2_j}) - \sum_{j=1}^{n} e'_j \mathrm{vlog}(\mathfrak{q}_j) = 0 \pmod{\ell}.$$

The unknown values of the linear system of equations formed by the relations are now these virtual logarithms.

If we want to be able to use the definition above, one must compute h_i, find the generators $\gamma_{\mathfrak{p}_j}$ and $\gamma_{\mathfrak{q}_j}$ and the set of generators of units of infinite order to get the exponents g_{i_j} for $i = 1, 2$. All these elements are computationally hard to obtain. The notion of Schirokauer maps now comes into play to overcome these difficulties.

Question 51. *What are Schirokauer maps?*

When selecting the polynomials f_i that define the number fields K_i, adding the condition $\ell \nmid \mathrm{disc}(f_i)$ implies that K_i has no roots of unity of order ℓ. This means that the isomorphism we have from Dirichlet's unit theorem implies the following isomorphism

$$\mathcal{O}_i^* / (\mathcal{O}_i^*)^\ell \cong (\mathbb{Z}/\ell\mathbb{Z})^{r_i}.$$

The goal of the Schirokauer map is to provide an explicit definition of this isomorphism.

Definition 22. *Let ℓ be a prime and K_i a number field with unit rank r_i. A Schirokauer map is a vector of homomorphisms defined as*

$$\Lambda_i := (\lambda_1, \lambda_2, \cdots, \lambda_{r_i}) : K_i^* / (K_i)^{*\ell} \to (\mathbb{Z}/\ell\mathbb{Z})^{r_i}.$$

with full rank on $\mathcal{O}_i^*$.

In order to construct the linear algebra system, we need the values $(e_1, \cdots, e_m, g_{1_1}, \cdots, g_{1_{r_1}})$ on one side and $(e'_1, \cdots, e'_n, g_{2_1}, \cdots, g_{2_{r_2}})$ on the other side. The coefficients e_j, e'_j are the valuation of the ideals $(a - b\alpha_i)$ at small prime ideals (the ones in the factor basis). The Schirokauer maps Λ_i are then defined such that the r_i coordinates g_{i_j} are precisely the coordinates of the Schirokauer map evaluated at the ideal. In other words, we want

$$\Lambda_i(a - b\alpha_i) = (g_{i_1}, g_{i_2}, \cdots, g_{i_{r_i}}),$$

for $i = 1, 2$. In order to obtain this, we make the following remark.

Remark 11. *The choice of fundamental units and the set of generators of the ideals of the factor basis is not unique.*

Let us consider fixed Schirokauer maps Λ_i for $i = 1, 2$. From the remark given just above, one can then appropriately choose the fundamental units and the generators such that

- $\Lambda_1(u_{1_j}) = (0, \cdots, 0, \underbrace{1}_{j^{th}}, 0, \cdots, 0)$ and $\Lambda_1(\gamma_{\mathfrak{p}_j}) = (0, 0, \cdots, 0)$.

- $\Lambda_2(u_{2_j}) = (0, \cdots, 0, \underbrace{1}_{j^{th}}, 0, \cdots, 0)$ and $\Lambda_2(\gamma_{\mathfrak{q}_j}) = (0, 0, \cdots, 0)$.

Then the evaluation of the Schirokauer map Λ_1 at the quantity $(a - b\alpha_1)$ gives

$$\Lambda_1(a - b\alpha_1) = \underbrace{\Lambda(\kappa_1)}_{=0} + \sum_{j=1}^{r_1} g_{1_j} \Lambda(u_{1_j}) + \sum_j e_j \underbrace{\Lambda(\gamma_{\mathfrak{p}_j})}_{=0} = (g_{1_1}, g_{1_2}, \cdots, g_{1_{r_1}})$$

as expected (and similarly for the other side). Finally, a relation will be expressed as

$$\sum_{j=1}^{r_1} g_{1_j} \mathrm{vlog}(u_{1_j}) + \sum_j e_j \mathrm{vlog}(\mathfrak{p}_j) - \sum_{j=1}^{r_2} g_{2_j} \mathrm{vlog}(u_{2_j}) - \sum_j e'_j \mathrm{vlog}(\mathfrak{q}_j) = 0 \pmod{\ell}$$

where g_{i_j} is the jth coordinate of Λ_i for $i = 1, 2$. The system of linear additive relations can be represented as in Table 4.2.

	$\mathrm{vlog}(\mathfrak{p}_1)$	$\cdots$	$\mathrm{vlog}(\mathfrak{q}_1)$	$\cdots$	$\mathrm{vlog}(u_{1_1})$	$\cdots$	$\mathrm{vlog}(u_{1_{r_1}})$	$\mathrm{vlog}(u_{2_1})$	$\cdots$	$\mathrm{vlog}(u_{2_{r_2}})$
$\mathrm{rel}^{(1)}$	$e_1^{(1)}$	$\cdots$	$e_1'^{(1)}$	$\cdots$	$g_{1_1}^{(1)}$	$\cdots$	$g_{1_{r_1}}^{(1)}$	$g_{2_1}^{(1)}$	$\cdots$	$g_{2_{r_2}}^{(1)}$
$\mathrm{rel}^{(2)}$	$e_1^{(2)}$	$\cdots$	$e_1'^{(2)}$	$\cdots$	$g_{1_1}^{(2)}$	$\cdots$	$g_{1_{r_1}}^{(2)}$	$g_{2_1}^{(2)}$	$\cdots$	$g_{2_{r_2}}^{(2)}$
$\vdots$										

$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{Schirokauer maps}}$$

Table 4.2: Representation of the matrix of relations used for the linear algebra step.

Concretely we now need two Schirokauer maps Λ_1 and Λ_2 each being a vector of r_i homomorphisms.

Question 52. *How are Schirokauer maps defined explicitly?*

Schirokauer proposed a map that satisfies the definition given above and moreover that is fast to evaluate. Recall that a Schirokauer map is a vector of homomorphisms defined as

$$\Lambda := (\lambda_1, \lambda_2, \cdots, \lambda_{r_i}) : K_i^* / (K_i)^{*\ell} \to (\mathbb{Z}/\ell\mathbb{Z})^{r_i}.$$

Let us consider an element $z \in K_i^*$ such that its denominator is coprime to ℓ. Moreover, let $f_i = \prod_j f_{i_j}$ (mod ℓ) assuming all the f_{i_j} are distinct. This is very likely when ℓ is large enough. By the Chinese Remainder Theorem, we have that

$$(\mathbb{Z}/\ell\mathbb{Z})[x]/(f_i) \cong \prod_j (\mathbb{Z}/\ell\mathbb{Z})[x]/(f_{i_j}).$$

Consider $X = \mathrm{lcm}(\ell^{\deg f_{i_j}} - 1)$. Then by Fermat's little theorem, we have $z^X \pmod{\ell} = 1 \in \mathbb{Z}/\ell\mathbb{Z}[x]/(f_i)$. Let us consider instead the quantity $z^X \pmod{\ell^2}$ in $(\mathbb{Z}/\ell^2\mathbb{Z})[x]/(f_i)$. Because $z^X - 1 \equiv 0 \pmod{\ell}$, we can re-write

$$z^X - 1 \equiv (0 \pmod{\ell}) \pmod{\ell^2}$$

and thus

$$z^X \equiv 1 + \ell S(z) \pmod{\ell^2}.$$

The map S in the equivalence above can be written as

$$S \quad : \quad K_i^*/(K_i^*)^\ell \quad \rightarrow \quad (\mathbb{Z}/\ell\mathbb{Z})[x]/(f_i)$$
$$z \quad \mapsto \quad \frac{z^X - 1}{\ell} \pmod{\ell} := S(z).$$

It is thus a polynomial in x with coefficients in $\mathbb{Z}/\ell\mathbb{Z}$ and can be expressed as $S(z) = s_0 + s_1 x + \cdots + s_{d-1}x^{d-1}$ with $s_i \in \mathbb{Z}/\ell\mathbb{Z}$, where d is the degree of f_i. Finally, we can compute the evaluation of the λ_i in $z \in K_i^*$ by choosing r_i coefficients of the polynomial S, i.e., we get $\lambda_i(z) = s_j \in \mathbb{Z}/\ell\mathbb{Z}$ (i and j not necessarily equal).

Remark 12. • *To compute the evaluation of the Λ_i, one can take any r_i coefficients among the $\deg f_i - 1$ ones of S or even r_i independent linear combinations of them.*

• *This map suggested by Schirokauer is the only known choice of Λ where the evaluation is efficient for any number field K_i.*

• *The definition of Λ_i depends on the representation of the number field K_i.*

> **Question 53.** *How can Schirokauer maps be used in the Tower setup?*

A rather simple trick allows us to consider Schirokauer maps in TNFS by using their definition from the non-tower context with a more "global" input number field.

More precisely, recall that a Schirokauer map is any morphism from $K_i^* \to (\mathbb{Z}/\ell\mathbb{Z})^{r_i}$ where K_i is a number field and r_i its unit rank. In the classical NFS setup, K_i is simply an extension of $\mathbb{Q}$, whereas in the Tower setup, K_i is an extension of $\mathbb{Q}(\iota)$.

It is then possible to define a Schirokauer map in TNFS by first defining an isomorphism from the intermediate fields $K_i = \mathbb{Q}(\iota, \alpha_i)$ to a number field K_{F_i} of degree $\deg h \times \deg f_i$ and then using a classical Schirokauer map $\Lambda_{\text{classical}}$ from the latter to $(\mathbb{Z}/\ell\mathbb{Z})^{r_i}$. In other words, we define the map Λ_i as

$$\Lambda_i : K_i^*/(K_i)^{*\ell} \xrightarrow{\cong} K_{F_i}^*/(K_{F_i})^{*\ell} \to (\mathbb{Z}/\ell\mathbb{Z})^{r_i}.$$

The field K_{F_i} is defined as $K_{F_i} = \mathbb{Q}[X]/(F_i)$ where F_i is a polynomial of degree $\deg h \times \deg f_i$. Such an isomorphism is not hard to find and will be detailed in Chapter 5. Note that since we have an isomorphism between K_i and K_{F_i}, the rank of units remains the same. Finally, the map

$$\Lambda_{\text{classical}} : K_{F_i}^*/(K_{F_i})^{*\ell} \to (\mathbb{Z}/\ell\mathbb{Z})^{r_i}$$

is simply a Schirokauer map as described above in the classical non-tower setup.

4.3 Focus on the relation collection

Now that we have presented the Tower Number Field Sieve algorithm in its generality, we focus on the relation collection step. Indeed, collecting relations in TNFS requires sieving in dimensions greater than 2 due to the shape of ϕ, in particular the number of coefficients involved. We start by introducing the special-q setup and then focus on the relation collection step.

4.3.1 The special-q setup

The relation collection phase looks at a set of linear polynomials $\phi(x, \iota) = a(\iota) - b(\iota)x \in \mathcal{R}[x]$ where a, b are polynomials of degree $\deg h - 1$ with $\deg h = \eta$ and bounded coefficients, and tries to identify which are going to produce doubly-smooth norms, i.e., for which pair $(a(\iota), b(\iota))$ we have $N_1(a(\iota) - b(\iota)\alpha_1)$ and $N_2(a(\iota) - b(\iota)\alpha_2)$ factor into small primes. To reduce the time of the sieving stage, Pollard [Pol93] suggested to divide the set of all polynomials ϕ, commonly called the search space, into multiple subsets. This task also requires a large amount of memory and Pollard's idea allows to parallelize the process, thus improving practical computation time. This corresponds to the so-called *special-q method*. This method regroups polynomials into groups such that $\phi(\alpha_1, \iota)$ (or $\phi(\alpha_2, \iota)$ depending on whether we put the special-q on the f_1-side or the f_2-side) share a common factor: the ideal $\mathfrak{Q}$. Thus, when talking about sieving algorithm, we usually consider a fixed special-q $\mathfrak{Q}$, and sieve the corresponding subset.

Let $\underline{\phi}$ denote the (row-) vector of coefficients of the polynomial $\phi(x, \iota)$, i.e., the vector

$$\underline{\phi} = (a_0, a_1, \cdots, a_{\eta-1}, b_0, b_1, \cdots, b_{\eta-1}) \in \mathbb{Z}^{2\eta}.$$

Question 54. *How to characterize divisibility by a prime ideal $\mathfrak{Q}$?*

Let us consider a special-q ideal $\mathfrak{Q}$ of degree 1 in K_i of the form

$$\mathfrak{Q} = \langle q, \iota - \rho_\iota, x - \rho_x \rangle,$$

where q is a prime number, ρ_ι is a root of h modulo q, and ρ_x is a root of f_i modulo q. One could also consider ideals of degree greater than 1, but special-q of degree 1 are the most common among ideals of bounded norms and thus we restrict to this case.

Proposition 2. *The $\mathfrak{Q}$-lattice $\mathcal{L}_{\mathfrak{Q}}$ is the set of polynomials ϕ such that the corresponding principal ideal in K_i is divisible by $\mathfrak{Q}$.*

Making Proposition 1 explicit allows to express the latter as follows.

$$\mathcal{L}_{\mathfrak{Q}} = \{(a_0, \cdots, a_{\eta-1}, b_0, \cdots, b_{\eta-1}) \in \mathbb{Z}^{2\eta} : \sum_{k=0}^{\eta-1} \left(a_k \iota^k - b_k \iota^k \alpha_i \right) \equiv 0 \pmod{\mathfrak{Q}}\}$$

where $i = 1, 2$ depending on the side we consider. A basis $B_{\mathfrak{Q}}$ of this lattice can be expressed as follows.

$$
\begin{array}{c}
(a,b) \\
(q,0) \\
(\iota - \rho_\iota, 0) \\
(\iota(\iota - \rho_\iota), 0) \\
\vdots \\
(\iota^{\eta-2}(\iota - \rho_\iota), 0) \\
(\rho_x, 1) \\
(\iota\rho_x, \iota) \\
\vdots \\
(\iota^{\eta-1}\rho_x, \iota^{\eta-1})
\end{array}
\begin{array}{ccccc|cccc}
a_0 & a_1 & \cdots & \cdots & a_{\eta-1} & b_0 & b_1 & \cdots & b_{\eta-1} \\
\left(\begin{array}{c} q \\ -\rho_\iota \\ 0 \\ \\ \\ \\ \rho_x \\ 0 \\ \\ \\ \end{array}\right. &
\begin{array}{c} 0 \\ 1 \\ -\rho_\iota \\ \\ \\ \\ 0 \\ \rho_x \\ \\ \\ \end{array} &
\begin{array}{c} \\ 0 \\ 1 \\ \ddots \\ \\ \\ 0 \\ \ddots \\ \end{array} &
\begin{array}{c} \\ \\ 0 \\ \ddots \\ -\rho_\iota \\ \\ \\ \\ \end{array} &
\begin{array}{c} 0 \\ \\ \\ \\ 1 \\ \\ \\ \\ \rho_x \end{array} &
\begin{array}{c} \\ \\ \\ \\ \\ 1 \\ \\ \\ \end{array} &
\begin{array}{c} \\ \\ \\ \\ \\ \\ 1 \\ \\ \end{array} &
\begin{array}{c} \\ \\ \\ \\ \\ \\ \ddots \\ \end{array} &
\left.\begin{array}{c} \\ \\ \\ \\ \\ \\ \\ \\ 1 \end{array}\right)
\end{array}
= B_{\mathfrak{Q}}.
$$

The determinant of this lattice is $q^{\deg \phi_h}$, where ϕ_h is an irreducible factor of $h \pmod p$. In our case $\phi_h = \iota - \rho_\iota$ because we only consider special-q ideals of degree 1 and so the determinant is simply q. The lattice dimension is 2η. For example, with $\eta = 3$, it is generated by the rows of the following matrix

$$
B_{\mathfrak{Q}} = \begin{pmatrix}
q & 0 & 0 & 0 & 0 & 0 \\
-\rho_\iota & 1 & 0 & 0 & 0 & 0 \\
0 & -\rho_\iota & 1 & 0 & 0 & 0 \\
\rho_x & 0 & 0 & 1 & 0 & 0 \\
0 & \rho_x & 0 & 0 & 1 & 0 \\
0 & 0 & \rho_x & 0 & 0 & 1
\end{pmatrix},
$$

where the first 3 columns correspond to the coefficients of $a(\iota) = a_0 + a_1\iota + a_2\iota^2$, and the last 3 columns to the coefficients of $b(\iota) = b_0 + b_1\iota + b_2\iota^2$. For example, the 5th row corresponds to the polynomial $\phi_5(x, \iota) = -\rho_x\iota + \iota x$, and $\underline{\phi}_5 = (0, \rho_x, 0, 0, 1, 0)$ is indeed in $\mathcal{L}_{\mathfrak{Q}}$. The lattice $\mathcal{L}_{\mathfrak{Q}}$ has dimension $d = 2\eta = 6$ and determinant q in this example.

Each unit of computation targets one special-q ideal $\mathfrak{Q}$ and searches for polynomials $\phi(x, \iota)$ with $\underline{\phi} \in \mathcal{L}_{\mathfrak{Q}}$ leading to relations, *i.e.*, for which both sides are smooth. In order to explore the lattice $\mathcal{L}_{\mathfrak{Q}}$, we first LLL-reduce the basis $B_{\mathfrak{Q}}$, and then consider linear combinations with small coefficients of these new basis elements. This allows us to concentrate on number field elements of small norms, thus increasing the probability of them being smooth. More precisely, let $M_{\mathfrak{Q}}$ be an LLL-reduced basis of $\mathcal{L}_{\mathfrak{Q}}$. We study the (row-) vectors $\mathbf{c}$ of coefficients such that

$$\underline{\phi} = \mathbf{c} \cdot M_{\mathfrak{Q}},$$

potentially leads to a relation. This is done using sieving algorithms.

4.3.2 Different algorithms for sieving

Constructing the double-divisibility lattice $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}}$

We focus on vectors $\mathbf{c}$ that belong to a sieving region $\mathcal{S}$. In NFS, the sieving region $\mathcal{S}$ is traditionally taken to be an ℓ_∞ ball, but in this work we will consider the ℓ_2 norm. Section 4.4.2 explains this preference.

   In order to efficiently detect the vectors $\mathbf{c}$ that give elements of smooth norms, one can perform an Eratosthenes-like sieving, marking quickly all vectors $\mathbf{c}$ in $\mathcal{S}$ leading to a norm on the f_1-side (or equivalently on the f_2-side) that is divisible by a small prime p. Repeating this sieve for many primes p allows to detect the most promising vectors $\underline{\phi}$, those for which the norm is divisible by many small primes. To do so, we proceed in a similar way as for the divisibility by $\mathfrak{Q}$.

   Let $\mathfrak{p}$ be a prime ideal of norm p in K_i of the form

$$\mathfrak{p} = \langle p, \iota - r_\iota, x - r_x \rangle,$$

where r_ι is a root of h modulo p and r_x is a root of f_i modulo p. The second statement of Proposition 1 can be reformulated for this specific context.

Proposition 3. *The principal ideal generated by $\phi(x, \iota)$ in K_i is divisible by $\mathfrak{p}$ if and only if $\phi(r_x, r_\iota) \equiv 0$ mod p.*

   Similarly as before, we translate this divisibility property for $\mathbf{c}$. Let $\mathbf{U}_\mathfrak{p}$ be the (row-) vector of size 2η corresponding to the modular equation of Proposition 3. More precisely, if we set

$$\mathbf{U}_\mathfrak{p} = \left(1, r_\iota, (r_\iota^2 \bmod p), \cdots, (r_\iota^{\eta-1} \bmod p), r_x, (r_x r_\iota \bmod p), (r_x r_\iota^2 \bmod p), \cdots, (r_x r_\iota^{\eta-1} \bmod p)\right),$$

the divisibility by $\mathfrak{p}$ is equivalent to the condition $\underline{\phi} \cdot \mathbf{U}_\mathfrak{p}^\mathsf{T} \equiv 0 \mod p$.

   Recall that the coefficients of the polynomials $\underline{\phi}$ are obtained from vectors $\mathbf{c}$ such that $\underline{\phi} = \mathbf{c} \cdot M_\mathfrak{Q}$. Thus taking into account the divisibility by $\mathfrak{Q}$, the condition becomes

$$\mathbf{c} \cdot M_\mathfrak{Q} \mathbf{U}_\mathfrak{p}^\mathsf{T} \equiv 0 \mod p,$$

and we are thus looking for the vectors $\underline{\phi} = \mathbf{c} \cdot M_\mathfrak{Q}$ with the above condition.

   The product $M_\mathfrak{Q} \mathbf{U}_\mathfrak{p}^\mathsf{T}$, reduced modulo p and normalized so that its first coordinate is 1, is expressed as

$$M_\mathfrak{Q} \mathbf{U}_\mathfrak{p}^\mathsf{T} \equiv \lambda \left(1, \alpha_1, \alpha_2, \ldots, \alpha_{\eta-1}\right)^\mathsf{T} \mod p,$$

with $\lambda > 0$. Since $M_\mathfrak{Q}$ and $\mathbf{U}_\mathfrak{p}$ are known, the values α_i can be explicitly computed. Note that this assumes the first coordinate is non-zero. Otherwise, one must either adapt the construction of $M_{\mathfrak{Q},\mathfrak{p}}$ below or skip the ideal during sieving. Finally, the set of vectors $\mathbf{c}$ such that $\mathbf{c} \cdot M_\mathfrak{Q} \mathbf{U}_\mathfrak{p}^\mathsf{T} \equiv 0 \mod p$ is represented by the vectors in the lattice $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}}$ generated by the rows of the matrix

$$M_{\mathfrak{Q},\mathfrak{p}} = \begin{pmatrix} p & 0 & 0 & 0 & \cdots & 0 \\ -\alpha_1 & 1 & 0 & 0 & \cdots & 0 \\ -\alpha_2 & 0 & \ddots & 0 & \cdots & 0 \\ \vdots & 0 & 0 & & \ddots & 0 \\ -\alpha_{\eta-1} & 0 & 0 & 0 & \cdots & 1 \end{pmatrix}.$$

In the end, since $M_{\mathfrak{Q},\mathfrak{p}}$ is explicitly known, we can compute the coefficients $\underline{\phi} = \mathbf{c} \cdot M_\mathfrak{Q}$ of the polynomials ϕ. This is possible as soon as we are able tto enumerate the short vectors $\mathbf{c}$ in the lattice above.

Cost of constructing $M_{\mathfrak{Q},\mathfrak{p}}$. The cost of LLL for full rank lattices is given by $O(d^6 \log^3(\max_i(||\mathbf{b}_i||_2)))$ where $\mathbf{b}_i$ are the basis vectors and $d = 2\eta$ is the dimension of the lattice. We assume the modular roots of the polynomials h, f_1 and f_2 are precomputed. Moreover, the cost of LLL-reducing the lattice generated by the basis $B_\mathfrak{Q}$ is done only once per special-q, thus we ignore it. Finally the cost of constructing the basis of the lattice $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}}$ narrows down to these operations:

- Constructing the vector $\mathbf{U}_\mathfrak{p}$: d operations; $d/2$ multiplications + $d/2$ modular reductions

- Compute the product $M_{\mathfrak{O}}\mathbf{U}_{\mathfrak{p}}$: $2d^2 - d$ operations: d^2 multiplications $+ d(d-1)$ additions.

- Normalization of the coefficients in the vector $M_{\mathfrak{O}}\mathbf{U}_{\mathfrak{p}}$: $d+1$ operations; 1 modular inverse $+ d$ multiplications.

- LLL-reduce the lattice generated by the basis $M_{\mathfrak{O},\mathfrak{p}}$: $\approx d^6 \log p$ operations.

In total we thus have $d^6 \log p + 2d^2 + d + 1$ operations to construct the basis $M_{\mathfrak{O},\mathfrak{p}}$.

It then remains to enumerate all the vectors in $\mathcal{L}_{\mathfrak{O},\mathfrak{p}}$ of bounded norms, and mark them to remember that the norms N_1 or N_2 are divisible by the prime p, which we also remember. Enumerating these vectors reduces to finding the set $\mathcal{L}_{\mathfrak{O},\mathfrak{p}} \cap \mathcal{S}$, where $\mathcal{S}$ is the sieving region. These vectors correspond to polynomials $\phi \in \mathcal{R}[x]$ that will lead to potential relations.

Different enumeration techniques exist in the literature which depend on the shape of $\mathcal{S}$ and the dimension d of the lattice in which we sieve. This dimension corresponds to the number of coefficients in $a(\iota), b(\iota)$ for TNFS and is usually equal to 2 for NFS since $a, b \in \mathbb{Z}$ (higher dimensions can also be considered for NFS). In two dimensions, thus for previous records using NFS, the sieving method of Franke and Kleinjung [FK05] is very efficient. However, in this chapter, we will focus on methods that can be used in higher dimensions.

Indeed, in TNFS, the relation collection phase considers vectors in a d-dimensional lattice with $d = \dim M_{\mathfrak{O},\mathfrak{p}}$. Taking the polynomials $a(\iota)$ and $b(\iota)$ of degree $\deg h - 1$ leads to $d = 2 \times \deg h$ hence $d \geq 4$. There exist two competitive methods in the literature that can be used when $d \geq 3$: the transition vectors method [Gré18] and the recursive hyperplane one [MR21]. We will detail both these algorithms in Section 4.4.1. These algorithms both use as a sieving space $\mathcal{S}$ a d-orthotope. In this chapter, we consider a new sieving space, a d-sphere. We justify our choice and describe our algorithm in Sections 4.4.2 and 4.4.3.

4.3.3 Other algorithms to find smooth norms

Other algorithms can be used to find relations by detecting whether the norms are doubly-smooth or not. To do so, these algorithms either find and extract smooth parts of the norms, or completely factors them. The family of batch algorithms and ECM are examples of such algorithms implemented and used in factorization and DLP computations.

Batch

Batch algorithms are quasi-linear time algorithms that test the smoothness of a list of integers. Multiple variants exist depending on the required output. The first two variants take as input the list of integers and a list of primes up to some bound: batch trial division outputs the list of primes p that divide the integers and batch smooth part extraction only returns the smooth part of the integers, *i.e.*, the product of the primes that divide the integers. Finally batch coprime factorization only takes as input the list of integers and returns the factors of each integer using repeated gcd operations.

In the context of DLP computations, the batch smooth part extraction algorithm is favored since it has the lowest complexity and is enough to test for potential smooth candidates. More precisely, the algorithm takes as input a list of prime numbers $P = \{p_1, \cdots, p_{\text{batch}}\}$ and a list of integers $N = \{n_1, \cdots, n_m\}$ and outputs for each $n_i \in N$ its smooth part, *i.e.*, the product of its prime factors in P. The algorithm uses product trees and remainder trees to efficiently compute the $\gcd(n_i, \prod_j p_j)$ for all i. We refer to [Ber08, Algorithm 2.1] for the complete algorithm.

The batch smooth part extraction has complexity $O(M(B) \log B)$ where $M(n)$ is the cost of multiplication of integers of size n bits and B is the number of bits of $\max\left(\prod p_j, \prod n_i\right)$.

Elliptic Curve Factorization Method

The elliptic curve factorization method [Len87], commonly known as ECM is an elliptic curve-based integer factorization algorithm that runs in sub-exponential time. The algorithm takes as input a composite integer n and outputs small factors of n. The algorithm relies on Hasse's theorem that states that if p is a prime, then the group order of an elliptic curve over $\mathbb{Z}/p\mathbb{Z}$ is $p+1-t$ where $|t| \leq 2\sqrt{p}$. The algorithm picks a random elliptic curve $E(\mathbb{Z}/n\mathbb{Z})$ and a point P on it and computes the scalar multiplication $Q = kP$

where $k = \prod p_i$ for primes p_i up to some bound. Let p be one of the factors of n which we want to find. If the order of the curve E over $\mathbb{Z}/p\mathbb{Z}$ divides k, then the z-coordinate of the point Q taken mod p is equal to zero and can thus be recovered by computing $\gcd(z, n) = p$.

The running-time of ECM is $O(L_p(1/2, \sqrt{2} + o(1))M(\log n))$, where p is the smallest prime factor of n.

4.3.4 Combining three algorithms

Sieving algorithms, batch and ECM are all algorithms used to find algebraic relations. These algorithms have different properties and in order to have the most efficient relation collection algorithm, one can combine them as a sequence of filters.

> **Question 55.** *How to best combine these three algorithms?*

These algorithms have different complexities and properties and thus cannot be used on the same amount of input norms N_i. Indeed, ECM is for example much more costly than sieving. Hence, applying it to the total amount of candidate norms N_i is far from optimal.

On the other hand, sieving is a much less costly algorithm per candidate, and thus can be used to find the small factors (up to a bound p_{max}) of a large number of structured candidates. This is why the relation collection step usually starts with a sieving algorithm with input all candidates $(a(\iota), b(\iota))$-pairs.

To summarize, sieving algorithms are used to find promising candidates by detecting $(a(\iota), b(\iota))$ pairs for which the corresponding norms N_i are divisible by many small primes. ECM is then used to guarantee that the norms of these promising candidates are indeed B-smooth by checking the larger prime factors. Batch smoothness can be added in between sieving and ECM or as a substitution of one of them to further optimize the overall cost. It is less costly than ECM and thus can be used to pre-select promising candidates but more costly than sieving and thus cannot be run on the entire set of candidates from the sieving area $\mathcal{S}$. It extracts prime factors up to a bound p_{batch} such that $p_{max} < p_{batch} < B$. The properties of these algorithms are described in Table 4.3.

Properties	Sieving	Batch	ECM
Input candidates	Numerous and structured	Numerous	Few
Prime factors extracted	Small	Small or medium	Large
RAM	Very large	Large	Tiny
Cost per candidate	Small	Medium	High

Table 4.3: Properties of the different relation collection algorithms

The relation collection can thus be seen as a sequence of filters, each taking a certain amount of candidates as inputs, and keeping "survivors" based on a criteria. These survivors are then the inputs to the next filter. The survivors are selected based on the size of the cofactor, which we now define.

> **Definition 23** ($\mathcal{A}$-cofactor). *Let N be a positive integer and consider $P = \prod_i p_i$ where the p_i are the prime factors of N extracted by Algorithm $\mathcal{A}$. Then the $\mathcal{A}$-cofactor is*
>
> $$C_{\mathcal{A}}(N) = \frac{N}{P}.$$

Remark 13. *When $\mathcal{A}$ is sieving or ECM, the algorithm returns the primes p_i and when $\mathcal{A}$ is batch smoothness, the algorithm directly returns the product P.*

For a fixed $\mathcal{A}$-cofactor bound $B_{\mathcal{A}}$, the survivors are the candidates selected if their norm N (or an approximation of it) satisfies $C_{\mathcal{A}}(N) \leq B_{\mathcal{A}}$. This entire procedure can be summarized in Figure 4.3.

Remark 14. *The bounds $B_{\mathcal{A}}$ are not smoothness bounds. However, they allow to select promising candidates. Indeed, if the cofactor of a norm is small, the probability of the norm being smooth is higher.*
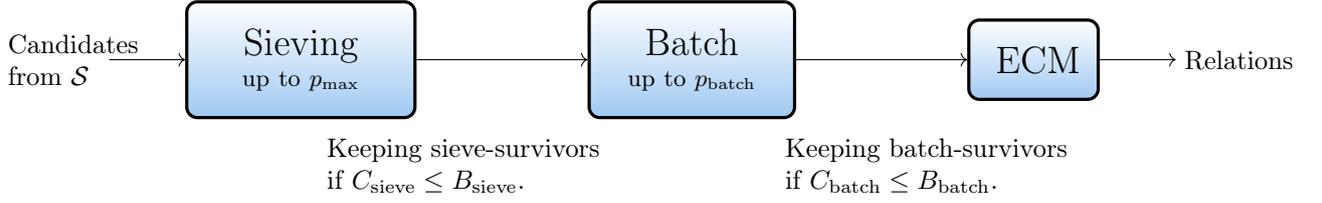
Figure 4.3: Sequence of algorithms for relation collection.

Finally, the complete relation collection algorithm is given by Algorithm 8.

Algorithm 8 Relation collection for a given special-q with sieving, batch and ECM

Input: A prime ideal $\mathfrak{Q}$, a sieving region $\mathcal{S}$
Output: A list of relations.

1: Construct the lattice $\mathcal{L}_{\mathfrak{Q}}$ and LLL-reduce it.
2: **for** each prime ideal $\mathfrak{p}$ in K_1 (or K_2) up to $p_{\max}$ **do**
3:     Construct the lattice $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}}$
4:     Enumerate all vectors in $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}} \cap \mathcal{S}$.
5:     For each vector enumerated, keep track of the size of the factors p with a sieving table.
6: For promising vectors, for which the product of the factors p is large, compute approximations of the norms N_1, N_2 and identify vectors with sieve-cofactor smaller than B_{siev}. They are called sieve-survivors.
7: Remove duplicates.
8: Run batch algorithm with input the norms N_1 and N_2 (this time computed exactly) of the sieve-survivors and primes up to p_{batch}. Keep batch-survivors whose batch-cofactor is smaller than B_{batch}.
9: Run ECM on the batch-survivors.
10: **return** Vectors with doubly-B-smooth norms which give relations as selected by ECM.

Technical details of Algorithm 8 will be given and illustrated in Chapter 5 with our concrete 521-bit computation. One can note that line 7 of Algorithm 8 removes duplicate relations. We now explain what they are and how we remove them.

4.3.5 Filtering through equivalent relations

When sieving through all the $(a(\iota), b(\iota))$-pairs it is sometimes the case that two pairs $(a(\iota), b(\iota))$ and $(a'(\iota), b'(\iota))$ will provide the same relation, *i.e.,* two linear equations that provide the same information on the virtual logarithms of the elements of the factor basis involved.

Identifying and removing duplicates is common in factoring and DLP computations. Let us start by identifying three different types of duplicates. Because this definition applies in both the classical NFS context and TNFS, we will use the terminology (a, b) to either define a classical $(a, b) \in \mathbb{Z}^2$ pair in NFS or $(a(\iota), b(\iota)) \in \mathcal{R}[x]$ in TNFS.

> **Definition 24** (Duplicates)**.** *A duplicate relation refers to a pair (a, b) such that there exists another pair (a', b') with $a' \neq a$ and $b' \neq b$ that leads to the same relation. We distinguish three types of duplicates:*
>
> - *We refer to **special-q-duplicates** when a relation with ideal factorization $(a + b\alpha_i)\mathcal{O}_i = \prod_i \mathfrak{p}_i^{e_i}$ involves several prime ideals $\mathfrak{p}_i$ that occur in the set of special-q's considered. In other words, more than one special-q unit computation provide the same relation.*
>
> - *If (a, b) generates a relation for a fixed special-q, then a K_h-**unit-duplicate** refers to the pair*

> (ua, ub) which gives the same relation for $u \in \mathcal{O}_{K_h}^*$, for u small enough.
>
> - Similarly, if (a, b) generates a relation for a fixed special-q, then a ζ_2-**duplicate** refers to the pair $(\lambda a, \lambda b)$ which gives the same relation for $\lambda \in \mathcal{O}_{K_h} \setminus \mathcal{O}_{K_h}^*$, for λ small enough.

Remark 15. *The special-q duplicates are considered over the entire set of relations, meaning for all special-q considered, whereas the other two types of duplicates concern a fixed special-q. The methods to remove them thus differ.*

We now take a closer look at duplicates and how they are dealt with in NFS and TNFS.

Question 56. *Why do we need to remove duplicates?*

Duplicate relations generate nearly identical lines in the matrix of the linear system of equations that needs to be solved in the linear algebra step. As the cost of solving the system is a function of its dimension, we want to get rid of all the unnecessary lines.

Moreover, obtaining a valid relation is costly. Indeed, as can be seen in Algorithm 8, outputting a valid relation implied computing norms, running batch smoothness and ECM algorithms over promising candidates, ... etc. In order to minimize the cost of these computations, the number of input candidates should be minimal and thus exclude all possibly identifiable duplicates.

Question 57. *When do we remove duplicates?*

In theory, a simple solution would be to remove duplicates just before solving the linear system, *i.e.,* during the filtering step. Indeed, the matrix is encoded as a list of relations represented as a list of prime ideal factors for each relation. It would then be enough to simply remove identical lines in this file before the linear algebra step.

However, in practice generating duplicate relations is costly as we mentioned before. More precisely, duplicates generate more hits during the enumeration of the vectors in $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}} \cap \mathcal{S}$ (line 4 in Algorithm 8), the size of the input to the batch algorithm is larger with duplicate relations (more sieve-survivors for which it was necessary to compute exact norms, line 8 in Algorithm 8) and this finally results in more batch-survivors and hence a more costly ECM algorithm (line 9 in Algorithm 8). It is thus convenient to get rid of the duplicates that can be identified as fast as possible. However, the same strategy cannot be applied for each type of duplicates.

Indeed, the special-q duplicates can only be detected once we know the factorization of the norms. Moreover, special-q computation units are often run in parallel and thus there is little hope to be able to detect any special-q duplicates before the end of the relation collection phase. These duplicates are thus removed during the filtering step.

On the other hand, K_h-unit-duplicates and ζ_2-duplicates are "local" to a special-q and can be detected at an earlier stage. For the latter, we thus have a trade-off between the cost of the different steps in the relation collection algorithm and the cost of analyzing whether a pair (a, b) yields a duplicate relation. In our Algorithm 8, we chose to remove duplicates before running the batch algorithm.

We now focus on the methods to remove K_h-unit-duplicates and ζ_2-duplicates during the relation collection step.

Question 58. *How do we identify and remove duplicates?*

A classical trick used in NFS is to reduce the search space by enforcing a positive sign to the first coordinate a. Indeed, when looking at K_h-unit-duplicates, we are concerned with elements $u \in \mathcal{O}_{K_h}^*$ and in the classical NFS setup, $\mathcal{O}_{K_h}^* = \mathbb{Z}^* = \{-1, 1\}$. By enforcing $a > 0$, we thus reduce the search space by a factor 2 and avoid all unit-duplicates.

The situation is more complicated in TNFS as the number of units is greater than 2. It is still possible to restrict to positive coefficients in order to avoid duplicates resulting from the units $\{\pm 1\}$ and we will see in Section 4.4 that the enumeration algorithm indeed only considers half of the vectors in $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}} \cap \mathcal{S}$. However, we are left with the following open question.

Open Question 2. *Is there a systematic way to identify and thus remove duplicates generated from units other than ± 1?*

The difficulty of answering this question comes not only from the large number of units that must be considered but also from the fact that the units must be small enough in order to produce a relation. Indeed, if u is too large with respect to the sizes of a and b, the factors in the norms of (ua, ub) will exceed the smoothness bound.

Before giving more details on how we deal with these problematic K_h-unit-duplicates and the ζ_2-duplicates, we give our methodology to identify them. We remind the reader that removing duplicates happens before the batch smoothness and thus we are looking at the set of sieve-survivors.

Our strategy to identify K_h-unit-duplicates and ζ_2-duplicates.

For each pair (a, b) resulting in a sieve-survivor, we compute the quantity

$$k := \frac{a}{b} \pmod{h} \in K_h,$$

and store the value k in a hash table. If (a, b) and (a', b') are either K_h-unit-dupicates or ζ_2-duplicates, then they will have the same index k. The hash table allows us to quickly identify if a given pair (a', b') is a duplicate of a previously seen (a, b) pair.

Remark 16. *This method is done "locally" for every special-q. Indeed, one could think of adapting the idea of a general hash table regrouping relations from every special-q considered but the memory cost would be exceedingly high.*

This method also justifies the choice of where in Algorithm 8 we test for duplicates. Indeed, computing $a/b \pmod{h}$ is not cost-free thus we want to avoid having to compute k for every pair (a, b) outputted by the enumeration algorithm. It is however less costly than computing an exact norm.

However, the method brings forth the following issue. Duplicates can be seen as an equivalence class for which we want to select a unique representative. This representative of the class should be the "smallest" pair (a, b), meaning the (a, b)-pair which leads to the smallest norms. The method given above does not necessarily keep the "smallest" pair. Indeed, if the pair $(\lambda a, \lambda b)$ for $\lambda \in \mathcal{O}_{K_h}$ is already in the hash table, then if the algorithm sees the pair (a, b) it will discard it and keep $(\lambda a, \lambda b)$.

Question 59. *Why do we want to keep the "smallest" pair?*

A larger (a, b)-pair adds non-zero coefficients in the matrix of the linear system of relations and thus slows down the linear algebra step. Indeed, considering the $(\lambda a, \lambda b)$-pair, we have

$$N_i(\lambda a, \lambda b) = N_i(a, b) N_{K_h}(\lambda)$$

for $i = 1, 2$ with the additional term $N_{K_h}(\lambda)$ with respect to the (a, b)-pair. This additional term yields extra ideals in the prime ideal decomposition, thus non-zero coefficients in the matrix. We illustrate this with a small example in NFS.

Example 3. *Let (a, b) and $(2a, 2b)$ be two pairs of integers that lead to the same relation. Then, looking at the f_1-side we have*

$$N_1(2a, 2b) = f_1(2b/2a)(2b)^{\deg f_1} = N_1(a, b) 2^{\deg f_1}.$$

We see that the additional $2^{\deg f_1}$ appears in the norm and thus prime ideals over 2 will appear in the factorization of the ideals and add non-zero coefficients to the matrix.

The removal of special-q duplicates is also easier when the representative of a duplicate class is in its canonic form. Indeed, recall that special-q duplicates are removed by simply comparing the lines in the file that encodes the relations. Thus if two different special-q's produce the same relation but each keep a different representative, say (a, b) for one and $(\lambda a, \lambda b)$ for the other, then their prime ideal decomposition will differ by some factors corresponding to $N_{K_h}(\lambda)$ and thus the duplicate will be kept.

Question 60. *How can one identify the "smallest" (a, b)-pair in a ζ_2-duplicates class of equivalence?*

The most intuitive idea is to consider the notion of a primitive pair.

Definition 25. *A pair (a, b) is primitive if there exists no $\lambda \in \mathcal{O}_{K_h} \setminus \mathcal{O}_{K_h}^*$ such that $a = \lambda a'$ and $b = \lambda b'$ with $a', b' \in \mathcal{O}_{K_h}$.*

In NFS, we will keep the (a, b)-pairs such that $\gcd(a, b) = 1$. The situation is more problematic in TNFS as the notion of gcd for $a(\iota)$ and $b(\iota)$ is not well-defined (the gcd exist at the level of ideals). We will thus detect non-primitive pairs by computing the gcd of their norms: if $\gcd(N_1(a, b), N_2(a, b)) = 1$, then the (a, b)-pair is primitive. Indeed if one considers $(\lambda a, \lambda b)$ which is clearly non-primitive, then we have

$$\gcd\left(N_1(\lambda a, \lambda b), N_2(\lambda a, \lambda b)\right) \geq N_{K_h}(\lambda)^{\min(\deg f_1, \deg f_2)} \neq 1.$$

Because we have stored and compared (a, b)-pairs in our hash table, we are left with a unique representative per class. However, as mentioned above, we cannot be certain that this pair is primitive. Indeed, the method that computes the indices k keeps the first pair it encounters with that index and ignores all the following regardless of their primitiveness. Hence, it remains to check whether our representative is primitive and if not, make it so.

After having removed the K_h-unit duplicates and ζ_2-duplicates as explained above, we thus run the following algorithm on each sieve-survivor.

Algorithm 9 Primitive representative for each class of duplicates

---

Input: (a, b)-pair corresponding to a sieve-survivor
Output: primitive (a, b)-pair corresponding to a sieve-survivor or Fail

1: Compute $\gcd(N_1(a, b), N_2(a, b))$
2: **if** $\gcd(N_1(a, b), N_2(a, b)) = 1$ **then**
3:      Return (a, b)-pair.
4: **else**
5:      **for** each prime $\ell | \gcd(N_1(a, b), N_2(a, b))$ **do**
6:          Try to find an element β in $\mathcal{O}_{K_h}$ of norm ℓ such that a/β and b/β are in $\mathcal{O}_{K_h}$.
7:          **if** Such a value β is found **then**
8:              $a \leftarrow a/\beta$ and $b \leftarrow b/\beta$
9:              Recompute $\gcd(N_1(a, b), N_2(a, b))$
10:             **if** $\gcd(N_1(a, b), N_2(a, b)) = 1$ **then**
11:                 Return new (a, b)-pair
12:             **else**
13:                 Move to next prime ℓ
14:          **else**
15:             Return Fail

Remark 17. *A few comments can be made on the above algorithm.*

- *In Algorithm 9 we use the fact that if $\gcd(N_1(a, b), N_2(a, b)) = 1$, then the (a, b)-pair is primitive. We actually have an equivalence if the number field K_h is principal. In particular, in our computation described in Chapter 5, the field K_h is principal which ensures we do not throw away too many relations by using Algorithm 9.*

- *The elements of K_h of norm ℓ are hardcoded in our code for our specific computation for primes ℓ up to 43.*

- *If for a given (a, b)-pair, once we have reached $\ell = 43$ and the gcd is still not 1, we simply remove the relation. Therefore the algorithm may still fail, even though K_h is principal.*

- *Finding the corresponding primitive pair can be done in an easier way with PARI or Sage by solving norm equations to find β in line 7. We avoid this in our code to minimize the dependencies to other libraries. Improving this algorithm and making it robust to non-principal K_h is left for future work.*

We have seen how to detect ζ_2-duplicates, keep a unique representative and make sure that the representative is in a primitive form. Unfortunately, Algorithm 9 does not work for finding a unique representative with respect to K_h-unit duplicates. Indeed, if γ is a unit, the quantity $N_{K_h}(\gamma)$ equals to 1. This naturally brings the following question.

Question 61. *How can one find the "smallest" (a, b)-pair in a K_h-unit-duplicates class of equivalence?*

We simply don't find a unique representative, and rely on the prime ideal decomposition which is unique in an equivalence class of K_h-unit duplicates.

4.4 Relation collection with lattice enumeration

In this work, we use an enumeration algorithm for Step 4 of Algorithm 8 where we consider a d-sphere as sieving region instead of a d-orthotope like existing methods. We use lattice enumeration techniques to efficiently obtain the set $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}} \cap \mathcal{S}$. We will use the notation $\mathcal{S} = S_d(R)$ to indicate we are working in a d-sphere of radius R or simply S_d to lighten the notation when possible.

4.4.1 Existing algorithms to enumerate $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}} \cap \mathcal{S}$

We start by giving a short refresher of two other recent competitive methods that can be used when $d \geq 3$: the transition vectors method [Gré18] and the recursive hyperplane one [MR21].

Transition vectors for lattice sieving in [Gré18]. Grémy suggests a sieving algorithm that is inspired by Franke-Kleinjung's algorithm in dimension 2 but can be extended to higher dimensions. Let $\mathcal{S}$ be the sieving space considered, in this case, a d-orthotope defined as the product of intervals $\mathcal{S} = [H_0^m, H_0^M[\times \cdots \times [H_{d-1}^m, H_{d-1}^M[$ for fixed bounds H_k^m, H_k^M.

The key notion used by Grémy to enumerate vectors of a lattice $\mathcal{L}$ in higher dimensions is the notion of transition-vectors, allowing to jump from vector to vector in order to reach all elements in $\mathcal{L}_{\mathcal{Q},\mathfrak{p}} \cap \mathcal{S}$. The transition-vectors are divided into d subsets $T_1, \cdots, T_d$, with T_k the set of k-transition-vectors for $k = 1, \cdots, d$. The latter have a non-zero k-coordinate and the last $d - k$ coordinates all equal to 0. The algorithm starts from $(0, 0, \cdots, 0) \in \mathcal{L}_{\mathfrak{Q},\mathfrak{p}}$ and enumerates all vectors in $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}} \cap \mathcal{S}$ by adding or subtracting transition-vectors. It starts with vectors of T_1 until it reaches the edges of $\mathcal{S}$, then looks at additions (or subtractions) of vectors of T_2 etc, increasing from 1 to d step by step.

More precisely, a k-transition vector $\mathbf{t}$ is a non-zero vector of the lattice $\mathcal{L}$ such that there exists vectors $\mathbf{v}$ and $\mathbf{v}'$ in $\mathcal{L} \cap \mathcal{S}$ satisfying $\mathbf{v}' = \mathbf{v} + \mathbf{t}$ with the following two conditions: the last $d - k$ coordinates of $\mathbf{v}$ and $\mathbf{v}'$ are the same, and the coordinate v_k' is the smallest possible value greater than v_k. The addition (similarly subtraction) of k transition-vectors is illustrated below.

$$
\begin{array}{rclccccccc}
\mathbf{v} + \mathbf{t} & = & (v_1, & v_2, & \cdots, & v_k, & v_{k+1}, & \cdots, & v_d) \\
& + & (*, & *, & \cdots, & *, & 0, & \cdots, & 0) \\
= \quad \mathbf{v}' & = & (v_1', & v_2', & \cdots, & v_k', & v_{k+1}, & \cdots, & v_d)
\end{array}
$$

We summarize the enumeration process in Algorithm 10 adapted from [Gré18]. In this description, we assume we are given a complete set of all transition-vectors $T = \{T_k\}_{k=1}^d$. For simplicity, we assume that all the intervals forming the sieving space are of the form $[-H, H]$ for some bound H. We denote the coordinates of the vector being enumerated as $\mathbf{v} = (v_1, v_2, \cdots, v_d)$. The algorithm takes as input a lattice $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}}$ of dimension d and a sieving region $\mathcal{S}$. It returns the list L of all elements in $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}} \cap \mathcal{S}$. The algorithm proceeds with recursive calls to the function $enum()$ for decreasing values of k.

Producing the entire set T is not possible in most cases, and thus the notion of transition-vectors is relaxed into nearly-transition-vectors, which serves a similar purpose. This variant is effective, albeit no longer reaches all vectors. A fall-back strategy is then considered when the algorithm fails to find an appropriate k-nearly-transition vector. Details are given in [Gré18].

Scaling up to higher dimensions. In dimension 4, this method seems to have sufficient prospects of success. However, even with the relaxed variant, experiments ran in dimension 6 in [Gré18] point to the

Algorithm 10 Simplified recursive enumeration algorithm from [Gré18]

Input: a lattice $\mathcal{L}$ of dimension d, a sieving region $\mathcal{S}$.
Output: list L of vectors in $\mathcal{L} \cap \mathcal{S}$

def enum($\mathbf{v} = (v_1, \cdots, v_d) \in \mathcal{S}$, k)
 1: $\mathbf{v}_{init} \leftarrow (v_1, \cdots, v_d)$
 2: **while** $v_k < H$ **do**
 3:   **if** $k > 1$ **then**
 4:     enum($\mathbf{v}$, $k - 1$)
 5:   Find $\mathbf{t} \leftarrow k$-transition vector $\in T_k$ such that $\mathbf{v} + \mathbf{t} \in \mathcal{S}$
 6:   $\mathbf{v} \leftarrow \mathbf{v} + \mathbf{t}$
 7:   Add $\mathbf{v}$ to L
 8: $\mathbf{v} \leftarrow \mathbf{v}_{init}$
 9: **while** $v_k \geq -H$ **do**
10:   **if** $k > 1$ **then**
11:     enum($\mathbf{v}$, $k - 1$)
12:   Find $\mathbf{t} \leftarrow k$-transition vector such that $\mathbf{v} - \mathbf{t} \in \mathcal{S}$
13:   $\mathbf{v} \leftarrow \mathbf{v} - \mathbf{t}$
14:   Add $\mathbf{v}$ to L
Run:
$L = \{(0, \cdots, 0)\}$
Get set $T = (T_1, T_2, \cdots, T_d)$ of transition vectors from oracle $\mathcal{O}(\mathcal{L}_{\mathfrak{Q}, \mathfrak{p}}, \mathcal{S})$.
Run $enum((0, 0, \cdots, 0), d)$
Return L.

---

limits of this method due to the poor quality of the nearly-transition-vectors and the number of calls required to the dedicated fall-back strategy. [Gré18] concluded that "*using cuboid search is probably a too hard constraint that implies the hardness or even an impossibility for the sieving process*".

Recursive lattice sieving through hyperplanes in [MR21]. McGuire and Robinson also proposed an enumeration algorithm in dimension 3 or higher. The sieving area being considered is a d-orthotope $\mathcal{S} = [0, H[\times [-H, H[\cdots \times [-H, H[$ for a fixed bound H. In dimension 3, it corresponds to a rectangular parallelepiped. The goal is to efficiently enumerate all the vectors in $\mathcal{L}_{\mathfrak{Q}, \mathfrak{p}} \cap \mathcal{S}$ as explained previously.

The main idea consists in dividing the search space into hyperplanes and enumerate in each of those hyperplanes. In order to obtain a fast sieving algorithm, the authors minimize the number of hyperplanes to visit by adequately choosing a "ground" hyperplane and then considering translations of it.

More precisely, in dimension 3 the "ground" plane G_0 is defined as a plane spanned by the two shortest vectors of $\mathcal{L}_{\mathfrak{Q}, \mathfrak{p}}$, namely $\mathbf{v}_1, \mathbf{v}_2$ through the origin, to which we subtract multiples of $\mathbf{v}_3$ until it does not intersect $\mathcal{S}$ anymore. Because of the small dimension of the lattice considered, these shortest vectors can easily be found using LLL. One then enumerates every point in $G_0 \cap \mathcal{S}$ before moving to the next translated plane: $G_1 = G_0 + \mathbf{v}_3, G_2 = G_0 + 2\mathbf{v}_3, \cdots, G_k = G_0 + k\mathbf{v}_3$ until a k is reached such that $G_k \cap \mathcal{S} = \emptyset$. For each translated plane, one enumerates points in $G_k \cap \mathcal{S}$.

> **Question 62.** *How to enumerate in a hyperplane G_k?*

The authors use integer linear programming to find a first point $p_0 \in G_k \cap \mathcal{S}$. Once p_0 is found (if it exists), the other points p_i are iteratively reached by adding multiples of $\mathbf{v}_1$ and $\mathbf{v}_2$, as long as points remain in $\mathcal{S}$.

Pseudo-code for dimension three is given in [MR21] for both the enumeration process and the linear programming step. We write in Algorithm 11 a pseudo-code of our understanding of how their method can be adapted in dimension d. The recursive function *enum*() takes as input a d-dimensional sieving space $\mathcal{S}$, and the LLL-reduced basis of the lattice $\mathcal{L}_{\mathfrak{Q}, \mathfrak{p}}$ of same dimension.

As we understand it, the McGuire and Robinson's short vectors serve a similar purpose as Grémy's

Algorithm 11 Recursive version of enumeration algorithm from [MR21]

Input: the basis of a lattice $\mathcal{L}$ of dimension d, a sieving region $\mathcal{S}$.
Output: list L of vectors in $\mathcal{L} \cap \mathcal{S}$

def enum(d, $\mathcal{S}$, $[b_1, b_2, \cdots, b_d]$)
 1: $L = \{\}$
 2: **if** $d \neq 1$ **then**
 3:      $k = 0$
 4:      $P = \text{plane}(\mathbf{0}, \mathbf{b}_1, \mathbf{b}_2, \cdots, \mathbf{b}_{d-1})$
 5:      $c_{\max} = \max\{c \in \mathbb{N} : \mathcal{S} \cap (P - c \cdot \mathbf{b}_d) \neq \emptyset\}$
 6:      $G_0 = P - c_{\max} \cdot \mathbf{b}_d$
 7:      **while** $G_k \cap \mathcal{S} \neq \emptyset$ **do**
 8:          $L' \leftarrow \text{enum}(d - 1, G_k \cap \mathcal{S}, [\mathbf{b}_1, \mathbf{b}_2, \cdots, \mathbf{b}_{d-1}])$
 9:          Append L' to L
10:          $k = k + 1$
11:          $G_{k+1} = G_k + \mathbf{b}_d$
12: **if** $d = 1$ **then**
13:      Find $p_0 \in \text{plane}(\mathbf{0}, \mathbf{b}_1) \cap \mathcal{S}$ with linear programming
14:      Add p_0 to L
15:      $c_{\max} = \max\{c \in \mathbb{N} : \mathcal{S} \cap (p_0 - c \cdot \mathbf{b}_1) \neq \emptyset\}$
16:      Define $P_0 = p_0 - c_{\max} \cdot \mathbf{b}_1$
17:      **while** $P_0 \cap S \neq \emptyset$ **do**
18:          $P_0 = P_0 + \mathbf{b}_1$
19:          Add P_0 to L
20: **return** L.

transition-vectors: namely, the aim is to choose relevant vectors to add (or subtract) to others while being as exhaustive in the search as possible. Similarly to Grémy's work, the enumeration here is not completely exhaustive. Indeed, in [MR21], the authors report consistently missing around 1.8 % of the lattice points per special-q due to corner cases.

> **Question 63.** *Why is this faster than [Gré18]?*

The main speed gain of this algorithm is not so much in the enumeration done in each translated hyperplane, but in the number of hyperplanes that are being considered. Indeed, taking the largest vectors of the basis for the direction of the hyperplane translations allows to minimize the number of hyperplanes required to cover $\mathcal{S}$.

Scaling up to higher dimensions. The authors only present their algorithm in dimension 3. Although they state that their algorithm can be extended to higher dimension, we wonder whether it remains efficient when $d \geq 3$. One difficulty we see is finding $c_{\max}$, which increases with the dimension d and can become too expensive very quickly. Indeed, finding $c_{\max}$ can be done using integer linear programming, which is doable in low dimension but should be very hard (or at least more costly than desired) as it grows.

4.4.2 Why do we choose a d-sphere?

Let d be the dimension of the lattice being considered. We explain here that as the dimension d of the sieving space increases, it is more efficient to choose a d-sphere.

Consider a d-sphere S_d and a d-orthotope C_d of equal volumes. The number of points to enumerate is thus the same if we consider $\mathcal{S}$ to be S_d or C_d. Let us assume that the size of the norms is only dependent on the size of the coordinates of the vector enumerated in $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}} \cap \mathcal{S}$. We will now argue that considering a d-sphere instead of a d-orthotope leads to smaller norms.

Recall that the volume of a d-sphere is given by
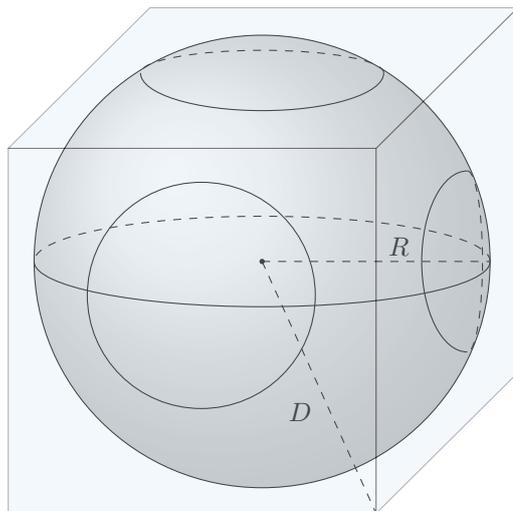
Figure 4.4: Hypercube and d-sphere for $d = 3$ of equal volume.

$$V_d(R) = \frac{\pi^{d/2} R^d}{\Gamma(d/2 + 1)},$$

and the volume of a d-hypercube of fixed length L is L^d. We use a d-hypercube instead of a d-orthotope to simplify the presentation. In order to have the same sieving volume, *i.e.*, $V_d(R) = L^d$ we must have

$$R = \frac{L \cdot \Gamma(d/2 + 1)^{1/d}}{\sqrt{\pi}}.$$

For the hypercube, the length of half the diagonal (from the center) is given by $D = \frac{L}{2} \cdot \sqrt{d}$. The distance between the summits of the hypercube and the d-sphere, the latter is expressed as

$$\frac{L}{2}\sqrt{d} - R = \frac{L}{2}\sqrt{d} - \frac{L \cdot \Gamma(d/2 + 1)^{1/d}}{\sqrt{\pi}},$$

and

$$\lim_{d \to \infty} \left(\frac{L}{2}\sqrt{d} - \frac{L \cdot \Gamma(d/2 + 1)^{1/d}}{\sqrt{\pi}} \right) = \infty.$$

Let $P_d = C_d \setminus S_d$ and $Q_d = S_d \setminus C_d$. Because we are considering S_d and C_d of equal volume, for $d \to \infty$, the quantity $D - R$ also tends to infinity as seen above. This quantity represents an upper bound on the distance from the origin to points in P_d, which would correspond to the largest norms. Hence, if we want to consider smaller norms, when $d \to \infty$ it is more advantageous to consider points in Q_d, and thus choosing a d-sphere as sieving area rather than a d-orthotope is a more suitable choice. This is illustrated in Figure 4.4.

4.4.3 Schnorr-Euchner's enumeration algorithm for TNFS

We now focus on the enumeration part, *i.e.*, the computation of $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}} \cap S_d(R)$. The algorithm follows Schnorr-Euchner's enumeration algorithm [SE94]. We have chosen to follow Schnorr-Euchner's enumeration strategy instead of the Fincke-Pohst-Kannan algorithm [FP85, Kan83] as it appeared more efficient operation-wise. We briefly describe Schnorr-Euchner's enumeration algorithm and refer to Chapter 2 for more details.

Description for TNFS

In order to find potential relations, one must enumerate all the vectors in $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}}$ of bounded norms. These vectors, which we denote $\underline{\phi}$, correspond to the coordinates of the polynomials $\phi(x, \iota)$, namely

$(a_0, \cdots, a_{\eta-1}, b_0, \cdots, b_{\eta-1})$. Recall that the lattice $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}}$ translates the notion of divisibility by an ideal $\mathfrak{p}$ and a special-q ideal $\mathfrak{Q}$. By enumerating vectors in $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}} \cap S_d(R)$ for many different $\mathfrak{p}$ (each generating a different $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}}$) one can identify vectors that are divisible by many $\mathfrak{p}$'s and thus more likely to correspond to B-smooth norms.

Let us fix $\mathfrak{p}$ and a special-q ideal $\mathcal{Q}$. The lattice $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}}$ is constructed as explained previously. Given an LLL-reduced basis $\{\mathbf{b}_1, \cdots, \mathbf{b}_d\}$ of $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}}$ and the radius R of a d-sphere S_d which corresponds to the sieving area, one can use Schnorr-Euchner's algorithm to find all these vectors.

Brief description of the algorithm. Recall that the algorithm constructs an enumeration tree where the leaves correspond to the vectors $\mathbf{c} = \sum_{i=1}^{d} v_i \mathbf{b}_i$ that satisfies $||\mathbf{c}||^2 \leq R^2$. Thus the leaves of the tree correspond to our vectors in $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}} \cap S_d$.

In order to search for these vectors, the enumeration method works in projected lattices and for a given level k in the tree, the projected vectors are expressed as

$$\pi_k(\mathbf{c}) = \sum_{j=1}^{d} \left((v_j + \sum_{i=j+1}^{d} (\mu_{i,j} v_i) \pi_k(\mathbf{b}_j^*) \right) = \sum_{j=k}^{d} \left((v_j + \sum_{i=j+1}^{d} (\mu_{i,j} v_i) \mathbf{b}_j^* \right),$$

where the vectors $\mathbf{b}_i^*$ correspond to the Gram-Schmidt orthogonalization of the basis vectors $\mathbf{b}_i$ and the $\mu_{i,j}$ are the Gram-Schmidt coefficients.

At each level k of the tree, the algorithm verifies that $||\pi_k(\mathbf{c})||^2 \leq R^2$ which can be reduced to enumerating admissible values of v_k that lie in a bounded interval. The algorithm visits only half the nodes since if $\mathbf{c} \in \mathcal{L}_{\mathfrak{Q},\mathfrak{p}}$ then $-\mathbf{c} \in \mathcal{L}_{\mathfrak{Q},\mathfrak{p}}$.

Efficiently computing the vectors $\mathbf{c} = \mathbf{v} \cdot M_{\mathfrak{Q},\mathfrak{p}}$. The algorithm works with the coefficient vectors $\mathbf{v} = (v_1, \cdots, v_d)$. However, in the end, we do not want the combinations $\mathbf{v}$, but the vectors $\mathbf{c} = \mathbf{v} \cdot M_{\mathfrak{Q},\mathfrak{p}} = \sum_{i=1}^{d} v_i \mathbf{b}_i$. Computing these vectors $\mathbf{c}$ can either be done naively, at the leaf level by explicitly computing $\mathbf{c} = \sum_{i=1}^{d} v_i \mathbf{b}_i$ for each leaf, or one can keep track of a partial sum $\sum_{i=t}^{d} v_i \mathbf{b}_i$ for a fixed value t chosen as input to the algorithm and update the quantity $v_i \mathbf{b}_i$ once a v_i is changed during the algorithm, *i.e.,* once the algorithm visits a new internal node in levels t to d. We opt for the second option as it reduces the overall cost of enumeration.

More precisely, let `common_part` $= \sum_{i=t}^{d} v_i \mathbf{b}_i$, where each $v_i \mathbf{b}_i$ is stored in a variable. Each time the algorithm visits a new internal node, thus updates v_i for a given $i = t, \cdots, d$, the algorithm updates `common_part` by subtracting the current $v_i \mathbf{b}_i$, computing the new $v_i \mathbf{b}_i$ with the new value of v_i and adding it back to `common_part`. Once at the leaf, in order to compute the vector $\mathbf{c}$, it remains to compute $\mathbf{c} = \sum_{i=1}^{t-1} v_i \mathbf{b}_i + $ `common_part`.

For most values of p we are concerned about, we use this optimized code with $t = 2$, thus updating all values $v_i \mathbf{b}_i$ during the algorithm, except at the leaf level, and finally computing $\mathbf{c} = v_1 \mathbf{b}_1 + $ `common_part`. When p becomes large and few leaves are found, it can be less efficient to choose $t = 2$, and thus one selects the appropriate $t > 2$ in order to optimize the number of operations performed for this computation. More details are given in the Section 4.4.4 below and the pseudo-code for the optimized enumeration algorithm is given in Algorithm 12.

Remark 18. *This optimization makes sense in this specific context where the lattices considered are of small dimension and often dense (in particular for small primes). This would not translate well for general lattices of larger dimensions or if only a handful of small vectors are outputted.*

4.4.4 Analysis of the enumeration algorithm

We now proceed to analyzing the enumeration algorithm. On the one hand, we want to estimate the cost of our enumeration algorithm. This implies having an estimate of the number of nodes and leaves in the enumeration tree. Moreover, this estimate is derived using the Gaussian heuristic. In order to do that, it is necessary to analyze the input lattice to the enumeration algorithm. Thus, we start by studying the behaviour of the lattice $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}}$.

The dimensions of the lattice we are considering are small, *i.e.,* precisely 6 in our example in Chapter 5 but plausible dimensions are 4, 6 or 8 for example depending on the degree of the polynomial h. Because

Algorithm 12 Optimized enumerating $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}} \cap S_d$

Input: LLL-reduced basis $\{\mathbf{b}_1, \cdots, \mathbf{b}_d\}$ of $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}}$, radius R of d-sphere S_d, variable t for optimization.

Output: List K of vectors $\mathbf{c} \in \mathcal{L}_{\mathfrak{Q},\mathfrak{p}} \cap S_d(R)$.

1: Pre-computation: compute all Gram-Schmit coefficients $\mu_{i,j}$ for $i < j$ and the norms of the Gram-Schmidt vectors $||\mathbf{b}_i^*||^2$ for all $i \leq d$.

2: $K \leftarrow \{\}$

3: $\sigma \leftarrow (0)_{(d+1) \times d}$, $r_0 = 0, r_1 = 1, \cdots, r_d = d$.

4: $\rho_1 = \rho_2 = \rho_{d+1} = 0$ $\triangleright$ with $\rho_k = ||\pi_k(\mathbf{c})||^2$

5: $v_1 = 1$, $v_2 = \cdots = v_d = 0$

6: $c_1 = \cdots = c_d = 0$ $\triangleright$ with $c_k = \sum_{i=k+1}^d \mu_{i,k} v_i$

7: $w_1 = \cdots = w_d = 0$

8: $\texttt{last\_nonzero} = 1$

9: $\texttt{common\_part} = v_t \mathbf{b}_t + \cdots + v_d \mathbf{b}_d$

10: $k = 1$

11: **while** true **do**

12:      $\rho_k = \rho_{k+1} + (v_k - c_k)^2 ||\mathbf{b}_k^*||^2$

13:      **if** $\rho_k \leq R^2$ **then**

14:          **if** $k = 1$ **then**

15:              $\mathbf{c} = \sum_{i=1}^{t-1} v_i \mathbf{b}_i + \texttt{common\_part}$ $\triangleright$ opt. computation of $\mathbf{c}$

16:              $K \leftarrow K \cup \mathbf{c}$

17:              **if** $\texttt{last\_nonzero} = 1$ **then**

18:                 Skip $\triangleright$ this generates ζ_2-duplicates

19:              **else**

20:                 **if** $v_k > c_k$ **then** $v_k \leftarrow v_k - w_k$

21:                 **else**

22:                    $v_k \leftarrow v_k + w_k$

23:                 $w_k \leftarrow w_k + 1$

24:          **else**

25:              $k \leftarrow k - 1$ $\triangleright$ we go down the tree

26:              $r_k \leftarrow \max(r_k, r_{k+1})$

27:              **for** $i = r_{k+1}$ to $k+2$ **do**

28:                 $\sigma_{i,k} \leftarrow \sigma_{i+1,k} + v_i \mu_{i,k}$

29:              $c_k \leftarrow -\sigma_{k+1,k}$

30:              $v_k = \lceil c_k \rfloor$, $w_k = 1$.

31:              **if** $k = \ell$ for $\ell = t, \cdots, d$ **then**

32:                 Re-compute $\texttt{common\_part}$ by updating $v_\ell \mathbf{b}_\ell$.

33:      **else**

34:          $k \leftarrow k + 1$ $\triangleright$ going back up the tree.

35:          **if** $k = d + 1$ **then**

36:              **return** K $\triangleright$ we find no more solutions

37:          $r_k \leftarrow k$

```
38:  if k ≥ last_nonzero then
39:      last_nonzero ← k
40:      v_k ← v_k + 1
41:      if k = ℓ for ℓ = t, ⋯, d then
42:          Re-compute common_part by updating v_ℓ b_ℓ.
43:  else
44:      if v_k > c_k then v_k ← v_k − w_k
45:          if k = ℓ for ℓ = t, ⋯, d then
46:              Re-compute common_part by updating v_ℓ b_ℓ.
47:      else
48:          v_k ← v_k + w_k
49:          if k = ℓ for ℓ = t, ⋯, n then
50:              Re-compute common_part by updating v_ℓ b_ℓ.
51:      w_k ← w_k + 1
```

of these small dimensions, we observed that classical analyzes of lattice reduction algorithms did not hold. For example, the following ratio $||\mathbf{b}_{i+1}^*||^2/||\mathbf{b}_i^*||^2 \geq \beta$ was observed in [NS06] for vectors outputted from a reduction algorithm. The constant β depends on the reduction algorithm considered and in the case of LLL, we have $\beta = 1/(\delta - \eta^2)$. Sage's default LLL implementation uses $\delta = 0.99$ and $\eta = 0.501$, thus $\beta = 1.35$. This value is obtained for random basis. Our lattices $\mathcal{L}_{\mathfrak{O},\mathfrak{p}}$ are however not random. We thus experimentally verified that for 6-dimensional lattices, the ratio $||\mathbf{b}_{i+1}^*||/||\mathbf{b}_i^*||$ is smaller than expected, hence we introduce the following heuristic.

Heuristic 1. *For 6-dimensional lattices $\mathcal{L}_{\mathfrak{O},\mathfrak{p}}$, the ratio $||\boldsymbol{b}_{i+1}^*||/||\boldsymbol{b}_i^*|| \approx 1.09$ on average.*

The result was compared to ratios obtained for random bases and bases of ideal lattices. Let us detail this now. In order to shed more light on the geometry of our $\mathcal{L}_{\mathfrak{O},\mathfrak{p}}$ lattices, we compare this ratio with the ratio obtained using two similar basis constructions: the Goldstein-Mayer bases, introduced in Chapter 2, and a constuction from [PS13], which generates cyclotomic ideal lattices.

Comparing to Goldstein-Mayer bases   Recall that a Goldstein-Mayer basis is given by the rows of the following matrix for a prime p and random elements $a_i < p$.

$$
\begin{pmatrix}
p & & & & \\
a_1 & 1 & & & \\
a_2 & & 1 & & \\
\vdots & & & \ddots & \\
a_{d-1} & & & & 1
\end{pmatrix}.
$$

For increasing values of p, we experimentally verify that for a 6-dimensional lattice, the ratio of norms $||\mathbf{b}_{i+1}^*||/||\mathbf{b}_i^*|| \approx 1.13$ on average with a standard deviation of 0.24. Recall that for our $M_{\mathfrak{O},\mathfrak{p}}$ bases, we had a ratio equal to 1.09 with the same standard deviation around 0.24. A statistical t-test confirms that the means of both the ratios from the Goldstein-Mayer bases and the $M_{\mathfrak{O},\mathfrak{p}}$ bases are significantly different.

Comparing to ideal lattices coming from [PS13].   Instead of using a random basis for a general lattice, we also look at the generator for ideal lattices proposed in [PS13] and used for the SVP ideal lattice challenge. The construction is similar to the Goldstein-Mayer lattices but instead of having a_i being randomly generated numbers in $[1, p]$, the coefficients of the first column are powers of a root of unity of a kth cyclotomic polynomial. In our case, we use $k = 7, 9, 14, 18$ in order to have a degree 6

polynomial. More precisely, the ideal lattice is generated by the rows of the following matrix

$$\begin{pmatrix} p & & & & \\ -a & 1 & & & \\ -a^2 & & 1 & & \\ \vdots & & & \ddots & \\ -a^{d-1} & & & & 1 \end{pmatrix}$$

with a the root mentioned above and p a prime we vary. Similarly as before, we average over many values of p and for different k-values that determine the cyclotomic polynomial. On average, we find that the ratio $||\mathbf{b}_{i+1}^*||/||\mathbf{b}_i^*|| \approx 1.06$ with a standard deviation of 0.046. A t-test confirms again that we reject the null hypothesis in favor of the alternative hypothesis that the difference in means is not equal to 0.

Going back to the analysis of the enumeration algorithm, we recall the following heuristic introduced in Chapter 2.

Gaussian heuristic. *For a given lattice $\mathcal{L}$ and a set $\mathcal{S}$, the number of points in $\mathcal{L} \cap \mathcal{S}$ is roughly the ratio of the volumes,* i.e., $\mathrm{vol}(\mathcal{S})/\mathrm{vol}(\mathcal{L})$.

This heuristic was suggested to analyze enumeration algorithms in [HS07] and experimentally confirmed to be accurate in [GNR10] for random lattices.

Number of leaves. The volume of a full-rank d-dimensional lattice $\mathcal{L}$ is given by $\det(\mathcal{L}) = \prod_{i=1}^{d} ||\mathbf{b}_i^*||$ and in our case the volume of $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}}$ is p. Using the Gaussian heuristic, the number of leaves is thus given by

$$\Xi_{\text{leaves}} = \frac{1}{2} \frac{\mathrm{vol}(S_d(R))}{\det(\mathcal{L})} = \frac{1}{2} \frac{R^d \pi^{d/2}}{\Gamma(d/2+1) \prod_{i=1}^{d} ||\mathbf{b}_i^*||} = \frac{R^d \pi^{d/2}}{2\Gamma(d/2+1)p}.$$

We experimentally verified for the dimension we are interested in, $d = 6$, that the number of leaves outputted by our enumeration algorithm is indeed very close to the Gaussian heuristic, see Chapter 5.

Number of nodes. Let Ξ_k denote the number of nodes at level k which corresponds to the number of points in $\pi_k(\mathcal{L}_{\mathfrak{Q},\mathfrak{p}}) \cap S_k(R)$. The volume of a projected lattice $\pi_k(\mathcal{L}_{\mathfrak{Q},\mathfrak{p}})$ is $\prod_{i=k}^{d} ||\mathbf{b}_i^*||$. Again, from the Gaussian heuristic we have

$$\Xi_k = |\pi_k(\mathcal{L}_{\mathfrak{Q},\mathfrak{p}}) \cap S_{d-k+1}(R)| = \frac{1}{2} \frac{\mathrm{vol}(S_{d-k+1}(R))}{\mathrm{vol}(\pi_k((\mathcal{L}_{\mathfrak{Q},\mathfrak{p}}))} = \frac{1}{2} \frac{R^{d-k+1} \pi^{(d-k+1)/2}}{\Gamma((d-k+1)/2+1) \prod_{i=k}^{d} ||\mathbf{b}_i^*||}.$$

The $\frac{1}{2}$ comes from the fact that we consider only half of the tree (see description above). We approximate the volume of the projected lattice $\pi_k(\mathcal{L}_{\mathfrak{Q},\mathfrak{p}})$ by $X \cdot p^{(d-k+1)/d}$, where X depends on Heuristic 1. We then get

$$\mathrm{vol}(\pi_k(\mathcal{L}_{\mathfrak{Q},\mathfrak{p}})) = \prod_{i=k}^{d} ||\mathbf{b}_i^*|| = ||\mathbf{b}_1||^{d-k+1}(1.09)^{\sum_{i=k-1}^{d-1} i} = ||\mathbf{b}_1||^{d-k+1}(1.09)^{\frac{1}{2}(d-k+1)(d+k-2)}.$$

Since for $k = 1$, we know that $\mathrm{vol}(\pi_1(\mathcal{L}_{\mathfrak{Q},\mathfrak{p}})) = p$, we can set $||\mathbf{b}_1|| = p^{1/d}/(1.09)^{(\sum_{i=1}^{d-1} i)/d}$. We then have

$$\mathrm{vol}(\pi_k(\mathcal{L}_{\mathfrak{Q},\mathfrak{p}})) = p^{(d-k+1)/d}(1.09)^{\sum_{i=k-1}^{d-1} i - ((d-k+1)/d) \sum_{i=1}^{d-1} i} = p^{(d-k+1)/d}(1.09)^{\frac{1}{2}(d-k+1)(k-1)},$$

and hence $X = (1.09)^{\frac{1}{2}(d-k+1)(k-1)}$. We get

$$\Xi_k = \frac{1}{2} \frac{R^{d-k+1} \pi^{(d-k+1)/2}}{\Gamma((d-k+1)/2+1)p^{(d-k+1)/d}(1.09)^{\frac{1}{2}(d-k+1)(k-1)}}.$$

Finally, the total number of nodes is $\Xi = \sum_{k=1}^{d} \Xi_k$. We experimentally verified this for typical 6-dimensional $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}}$ lattices for varying p, see Chapter 5.

Running time of enumeration. The running time of the enumeration algorithm is given by the number of nodes Ξ times the number of operations per node. At each node, the algorithm performs 7 arithmetic operations on average to compute and update the linear combinations $\mathbf{v}$. In addition to looking for the combinations $\mathbf{v}$, one must also compute the vector $\mathbf{c} = \mathbf{v} \cdot M_{\mathfrak{Q},\mathfrak{p}} = \sum_{i=1}^{d} v_i \mathbf{b}_i$. As mentioned above, this can either be done naively at the leaf level by explicitly computing $\mathbf{c} = \sum_{i=1}^{d} v_i \mathbf{b}_i$ for each leaf, which costs $2d^2 - 1$ extra operations per leaf.

Or, one uses $\texttt{common\_part} = \sum_{i=t}^{d} v_i \mathbf{b}_i$. Each time the algorithm visits a new internal node in the levels t up to d, thus updates v_i for a given $i = t, \cdots, d$, the algorithm performs $4d - 1$ operations: in order to update $\texttt{common\_part}$, we subtract the current $v_i \mathbf{b}_i$ (d operations), compute the new $v_i \mathbf{b}_i$ ($2d-1$ operations) with the new value of v_i and add it back to $\texttt{common\_part}$ (again, d operations).

Once at the leaf, in order to compute the vector $\mathbf{c}$, it remains to perform $(t-1)(2d-1) + t - 1$ operations, $\mathbf{c} = v_1 \mathbf{b}_1 + \cdots + v_{t-1} \mathbf{b}_{t-1} + \texttt{common\_part}$. In summary, we have for the additional cost of computing the vector $\mathbf{c}$

$$\texttt{Comp c naively} = \#\text{ leaves} \times (2d^2 - 1) = \Xi_1(2d^2 - 1),$$

and using $\texttt{common\_part}$,

$$\texttt{Comp c opt} = (\#\text{ int. nodes}_{t \to d} \times (4d - 1)) + (\#\text{ leaves} \times ((t-1)(2d-1) + t - 1))$$

$$= \left(\sum_{i=t}^{d} \Xi_i \right) (4d - 1) + \Xi_1((t-1)(2d-1) + t - 1).$$

We experimentally verified that the optimized code results in less operations than the naive one to compute all the vectors c for all but too large values of p when choosing $t = 2$. When p becomes too large and there aren't many leaves, the optimized code uses more operations than the naive one. One easy way to resolve this is to increase the value of t in the definition of $\texttt{common\_part}$. However, this occurs when p is large enough that the predominant cost is in generating the lattice $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}}$ and not in the enumeration algorithm, see Figure 4.5. Finally, the total cost of enumeration on average is thus equal to

$$\texttt{Cost enum} = 7 \times \Xi + \texttt{Comp c opt}.$$

Number of leaves per node. Finally, the number of leaves per node is given by

$$\frac{\#\text{ leaves}}{\#\text{ nodes}} = \frac{\Xi_1}{\sum_{k=1}^{d} \Xi_k},$$

as a function of p. When p is small, the number of leaves per node is high and decreases with p. Indeed, the probability of a norm being divisible by a small prime is higher than for larger primes. Hence for small primes, the ratio of leaves with respect to the total number of nodes visited by the enumeration algorithm is close to 1. This is illustrated in Chapter 5, Figure 5.1 for parameters specific to our computation.

Comparing enumeration and constructing $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}}$. Figure 4.5 illustrates the variation of the number of operations for the construction of the basis $M_{\mathfrak{Q},\mathfrak{p}}$ for the lattice $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}}$ and the enumeration algorithm for increasing values of p and a fixed special-q. We clearly see that when p is small, the enumeration algorithm is more costly (in terms of number of operations). However, when p becomes large enough, constructing the basis $M_{\mathfrak{Q},\mathfrak{p}}$ becomes much more costly. The intersection point varies depending on the radius R and can be chosen to be close to $p_{\max}$.

4.4.5 Overall complexity of relation collection

The total cost of Algorithm 8 is the sum of the cost of constructing the lattice $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}}$, the cost of enumerating in $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}} \cap S_d(R)$, and the costs of batch smoothness on the sieve-survivors and ECM on the batch-survivors. In order to optimize the overall complexity, it is important to correctly set the many parameters that come into play during this step. In particular, one must decide:

1. the size of (many) fixed parameters: the radius R, the smoothness bound B, the range of special-q's to consider, the bounds $p_{\max}, p_{\text{batch}}$.
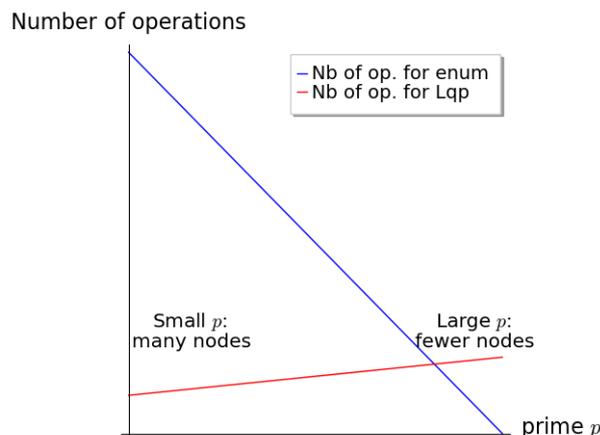
Figure 4.5: Number of operations for enumeration and constructing a basis of $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}}$ as a function of p. This plot illustrates the behaviour of these two costs.

2. the balance between sieving, batch smoothness and ECM based on the size of the cofactors.

A detailed analysis of the effect of the parameters R, B and the range of special-q's on the total cost of the relation collection will be given in Chapter 5, Section 5.3.1. More generally, we will discuss our choices for these parameters in Chapter 5 as they mostly come from experimental observations and testing.

4.5 Comparing with other methods

4.5.1 Comparing with [Gré18]

- Grémy's algorithm uses a d-orthotope as sieving space, whereas we consider a d-sphere. As explained previously, we believe that as the dimension increases, it is more efficient to sieve in a d-sphere as opposed to a d-orthotope. We send the reader back to Section 4.4.2 for a more detailed explanation.

- As the dimension increases, so does the number of nearly-transition vectors required in [Gré18] for the algorithm to enumerate most of the vectors. These nearly-transition vectors are generated during the initialization of the enumeration procedure using various strategies.

- Moreover, [Gré18] indicates that in dimension 6, the number of calls to the fall-back strategy is important, indicating that the nearly-transition-vectors are of poor quality, and thus the algorithm requires the use of skew-small-vectors (also to be computed).

- Finally, Grémy's algorithm is not exhaustive in its search of vectors in $\mathcal{L}_{\mathcal{Q},\mathfrak{p}} \cap \mathcal{S}$, and as the dimension increases, in addition to what was mentioned just before, we suspect the percentage of missing vectors increases.

4.5.2 Comparing with [MR21]

- Similarly as Grémy's algorithm, this algorithm also uses a d-orthotope as sieving space. We send the reader back to Section 4.4.2 for a more detailed argumentation.

- The algorithm presented in [MR21] is very similar to the classical enumeration algorithm of Fincke-Pohst-Kannan (FPK) [FP85, Kan83] adapted to a rectangular sieving region. One important cost in both FPK and this algorithm is finding the initial point in each plane from which the enumeration starts. In [MR21], this is done with linear programming. Every time the algorithm changes hyperplane, an integer linear programming problem must be solved. This does not add much complexity to the algorithm, but its cost is non-negligible with respect to the rest of the operations performed, and increases with the dimension. Our algorithm is based on Schnorr-Euchner's variant

which starts its enumeration of a given interval at its center. This avoids the computation of the edge of the interval at each level as required in FPK or similarly the linear programming cost.

- Moreover, as the dimension grows, so will the number of hyperplanes. Thus, we believe that the algorithm will struggle to be competitive when the number of hyperplanes becomes too important and a linear program must be solved for each hyperplane.

- Finally, our algorithm is exhaustive by construction, and thus enumerates every single vector in $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}} \cap S_d$. As mentioned previously, the algorithm in [MR21] encounters boundary issues when the planes intersect only the corners of the sieving region. The loss is reasonable in dimension 3 but may become more and more problematic as the dimension grows.

4.6 Conclusion

In this chapter, we described the overall setup of the TNFS algorithm with a particular emphasis on the relation collection step. Indeed, this step differs quite significantly from the classical NFS setup as it requires sieving in higher dimensions than for the latter. We thus proposed a new approach to sieving by not only considering the ℓ_2 norm instead of the previously used ℓ_∞ norm, but also by adapting the known Schnorr-Euchner enumeration algorithm to this context. This new sieving strategy along with the clarification of technical details such as Schirokauer maps and duplicates allowed us to successfully implement the TNFS algorithm and perform a discrete logarithm computation with it. Details about the implementation and the computation are given in the following chapter.

Chapter 5

An implementation and a 521-bit $\mathbb{F}_{p^6}$ record with TNFS

This chapter describes the first ever implementation of the Tower Number Field Sieve algorithm as well as the computational details of our discrete logarithm computation in a finite field of 521-bit. Whereas the theoretical descriptions of the various steps of the algorithm were given in Chapter 4, we focus here on the technicalities linked to the implementation and the record. We describe the results obtained at the various stages of the algorithm as well as their execution time. The total computation took 2.8 core years on a machine with two CPUs Intel Xeon Gold 6130 with 192 GB of RAM. The computation was performed on the Grid5000 cluster.

In addition to a description of the record, we compare the latter with computations performed with the classical NFS algorithm for both factoring an integer of the same size and DLP over the same size finite field. We also compare it with NFS computations with sieving in higher dimensions as done in [GGMT17] and [MR21]. Finally, we discuss the limits of this algorithm with respect to increasing sizes of finite fields and difficulties foreseen for much larger computations.

Contents

5.1 Our target

We consider a 521-bit finite field $\mathbb{F}_{p^n}$ where $p = $ `0x6fb96ccdf61c1ea3582e57` is a 87-bit prime and $n = 6$. The extension degree n is composite with factors $\eta = 3$ and $\kappa = 2$. The prime p is chosen to be the closest prime to the 87 first bits of RSA-1024, the 1024-bit integer coming from the RSA Factoring Challenge. Moreover, we chose as target element in $\mathbb{F}_{p^6}$ an element whose decimals are taken from π, that is

$$\texttt{target} = (31415926535897932384626433 + 83279502884197169399375105\iota$$
$$+ 82097494459230781640628620\iota^2) + x(89986280348253421170679821$$
$$+ 48086513282306647093844609\iota + 55058223172535940812848111\iota^2)$$

Note that the values of p and of the target are not chosen by us but taken from an outside source which allows to reinforce the authenticity of our computation.

Now recall from Chapter 1 that the computation of a discrete logarithm in a group can be reduced to its computation in one of its prime subgroups by Pohlig-Hellman's reduction.

> **Question 64.** *How is this subgroup chosen?*

We want to consider a prime order subgroup for which the computation is the hardest. To do so, we will work modulo $\ell = p^2 - p + 1 = $ `30c252a90b588491be0a93f6fd11924531a80adb333b`, the 174-bit prime order of the 6th cyclotomic subgroup of the multiplicative group of our target finite field. Indeed, any other factor of $p^6 - 1$, namely $p - 1, p + 1$ and $p^2 + p + 1$, correspond to subfields in which DLP would be much easier to solve. For example, if ℓ divides $p - 1$, the elements considered in DLP belong to $\mathbb{F}_p^*$ and solving DLP with NFS in this prime field of 87 bits can be done in nearly negligible time. On the other hand, the value $p^2 - p + 1$ does not correspond to any subfields of $\mathbb{F}_{p^6}$.

> **Question 65.** *Why did we choose this target?*

The choice of the finite field, more precisely the extension degree, was motivated by the use of such finite fields in pairing-based protocols. Recall that a pairing takes as inputs points on an elliptic curve defined over a finite field $\mathbb{F}_{p^n}$. As already discussed in Chapter 3, the security of pairings comes from the balance between the hardness of DLP on the elliptic curve side and the finite field side.

In [FST10], the authors introduce the notion of *pairing-friendly* curves, meaning elliptic curves with small embedding degree n and a large prime-order subgroup. It is further mentioned that to reach a 128-bit level of security, one must consider an extension field of bitsize around 4000 with $n \in [6, 10]$ if $\rho = 2$ and $n \in [12, 20]$ if $\rho = 1$. The authors then propose a family of (supersingular) elliptic curves [FST10, Section 3.3] for which the embedding degree is equal to 6, and thus the target group $\mathbb{G}_T$ of the pairing is $\mathbb{F}_{p^6}^*$ for a prime p. The elliptic curves MNT (with embedding degree 6) are examples of such curves.

For a more recent analysis of pairing-friendly curves, we refer to Guillevic's blogpost [Gui]. While for current levels of security we see that n is strictly greater than 6, for ZK-snarks with cycles of curves, [1] MNT-6 curves are widely deployed.

We further chose a finite field size for which a DLP computation seemed possible in reasonable time and we discuss in Section 5.7 the possibility of extending our computation to larger finite fields.

5.2 Polynomial selection

As explained in Chapter 4, three polynomials with specific characteristics must be chosen for TNFS. The following polynomials were provided to us by Guillevic after careful consideration of which polynomial selection method would provide the best polynomials. As explained in Chapter 4, in order to choose between the Conjugation method or JLSV1, it suffices to compare the size of the norms N_1 and N_2 for many $(a(\iota), b(\iota))$-pairs. Indeed, the parameters of f_2 outputted by Conjugation are the same as the parameters of f_1 and f_2 outputted by JLSV1 and thus it suffices to compare the norms N_1, N_2 for the Conjugation method. Averaging over 10^5 $(a(\iota), b(\iota))$-pairs clearly showed that N_1 was much smaller

---

[1] A cycle of curves (currently only available with MNT curves) is a pair of pairing-friendly elliptic curves $\mathcal{E}_1, \mathcal{E}_2$ such that $\mathcal{E}_1$ is defined over a finite prime field $\mathbb{F}_p$ with prime order r, and $\mathcal{E}_2$ is defined over the finite field $\mathbb{F}_r$ with order p.

than N_2, and thus the Conjugation method was finally considered to choose the polynomials. Code to reproduce these results can be found at https://gitlab.inria.fr/tnfs-alpha/alpha.git.

The choice for h. The polynomial h is of degree $\eta = 3$, monic and irreducible modulo p. In our computation, we use

$$h(\iota) = \iota^3 - \iota + 1.$$

This polynomial has the following property: $1/\zeta_2(K_h) = 0.9009$. We could have chosen a polynomial h of degree $\eta = 2$. This would have resulted in a sieving space of dimension 4 instead of 6. However, for a first large-scale experiment, the goal was to explore the efficiency of sieving in higher dimension, thus the choice of a degree 3 polynomial. We leave for future work the comparison of a TNFS computation with a polynomial h of degree 2.

The choices for f_i. The polynomials f_1, f_2 were selected using the Conjugation method. We recall that this method looks for polynomials of degree κ and 2κ. We use

$$f_1 = x^4 + 1,$$

and

$$f_2 = 11672244015875x^2 + 1532885840586x + 11672244015875.$$

These polynomials are irreducible over $\mathbb{Z}[\iota][x]$ and the polynomial f_2 corresponds to the irreducible common factor of these two polynomials, as it is the case for the Conjugation method. As mentioned in Chapter 4, additional criteria can be taken into account when selecting these polynomials such as Murphy's α value [GS21].

5.3 Collecting relations

The algorithm to collect relations in TNFS is described in Algorithm 8 in Chapter 4. We now focus on specific details related to our computation and our implementation.

5.3.1 Adjusting parameters before sieving

Before starting to collect relations one must fix the many parameters that come into play during Algorithm 8. The computational cost of sieving greatly varies depending on the balance between these parameters. In particular, one needs to fix a smoothness bound B, a range of special-q's to consider and a sieving space, which in our context reduces to choosing the radius R of the 6-dimensional sphere we sieve with.

Getting enough relations

In order for the linear algebra step of the algorithm to succeed, one must collect enough relations to construct the linear system of equations. The smoothness bound B gives the number of relations that are needed for the linear algebra step to succeed, *i.e.*, the required number of relations N_{tot} corresponds to $2\pi(B)$, where $\pi(x)$ is the prime counting function. Indeed, the factor basis regroups prime ideals of $\mathcal{O}_1$ and $\mathcal{O}_2$ of norm less than B and of degree 1 and thus the size of the factor basis is twice the number of prime ideals of norm less than B. By Chebotarev density theorem, the latter is given by the counting function π. A good approximation for the counting function is the logarithmic integral function.

Interestingly enough, two opposite considerations can be made. In theory, we would want slightly more relations in order to obtain a full-rank matrix. In practice, we require slightly less relations since some ideals will not be considered in the factorizations of the norms.

Now that we know an approximation of how many relations we need, one must balance the range of special-q's, the smoothness bound B and the radius R of the sieving space in order to optimize the total cost of the sieving step. Let $[q_{\min}, q_{\max}]$ denote the range of special-q to use for sieving.

In order to choose this interval in an optimal way, one must first estimate how many relations are generated on average from a single special-q of a given size. To do so, one can use the Dickman ρ function to estimate the probability of smoothness for the norms N_1 and N_2, as explained in Chapter 3. This allows us to get an estimate of the number of relations expected from a special-q as a function of B. Using this estimation, one can then balance the parameters (q_{min}, R, B) in order to find the optimal combination. We defined the total cost (of relation collection) in this context to be the number of special-q required

times the computation time for a single special-q. At this early stage, since we are looking at estimations, the time of sieving for a single special-q is taken to be R^6. The volume of a 6-sphere is given by $\frac{\pi^3}{\Gamma(4)} \cdot R^6$ and we ignore the constants at this stage. These parameters influence the total cost of sieving in the following way.

Impact of R: When choosing a large radius R, one expects to find more relations per special-q since the sieving space is larger. However, increasing R also increases the cost of enumeration for a given special-q. Overall for a fixed q_{min} and B, we observed in our experiments that when R increases, the total cost increases too. Indeed, when R increases the number of special-q needed decreases (which is both good and expected). However, this is counter-balanced by the increase in the cost of enumeration.

Impact of B: Similarly, increasing the smoothness bound B naturally increases the probability of being smooth. This results in more expected relations per special-q and thus a smaller interval would be required. On the other hand, the total number of relations required is equal to $2\pi(B)$ relations and thus the higher the value of B, the more relations are needed. This can be observed in our experiments where for increasing B, the total cost decreases and then starts increasing again.

Impact of q_{min}: The larger the special-q, the less relations it is expected to produce and thus we will require a larger interval. But if $q_{\min}$ is too small, one must take into account the effect of the special-q duplicates, see Chapter 4, Definition 24.

The behaviour of the parameters are summarized in Table 5.1 where $\nearrow$ means the parameters is being increased and $\searrow$ decreased.

parameters	# relations required	# special-qs	total cost
q_{min} $\nearrow$	–	$\nearrow$	$\nearrow$
R $\nearrow$	–	$\searrow$	$\nearrow$
B $\nearrow$	$\nearrow$	$\searrow$ then $\nearrow$	$\searrow$ then $\nearrow$

Table 5.1: Influence of increasing the size of parameters

In our computation, we chose the parameters

$$q_{min} = 5,000,113 \approx 2^{22.2}, \quad q_{\max} = 26,087,683 \approx 2^{24.6}, \quad B = 2^{27}, \quad R = 21.$$

This results in a total of 1,280,000 special-q's. Further optimizing these parameters with more extensive experiments is left for future work.

5.3.2 Analyzing the sieving step: enumerating in a lattice

The first step in collecting relations is to run a sieving algorithm to collect promising relations. This is done using Algorithm 12 from Chapter 4 which enumerates all vectors in $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}} \cap S_6(R)$ for a radius R and a range of special-q values determined above. In addition to these parameters, one must also select a prime bound $p_{\max}$ and a cofactor bound B_{sieve}.

In our code, enumeration is done with the program `enum.c` which takes as input a radius R, a cofactor bound B_{siev} and pre-computed files containing roots of the polynomials $h, f_1, f_2 \pmod{p}$ for primes $p \le p_{\max}$ and a range of special-q with the associated reduced bases $M_{\mathfrak{Q}}$. It outputs the list of sieve-survivors.

Question 66. *On which side do we consider the special-q's?*

Recall that the special-q method regroups polynomials ϕ such that $\phi(\alpha_i, \iota)$ share a common factor in K_i, the ideal $\mathfrak{Q}$. Considering this common factor can be done for either the f_1-side or the f_2-side. It is common to consider the special-q on the side which produces the largest norms since it will be harder to find B-smooth norms for it. In our case, the polynomial f_2 leads to much larger norms and thus we consider the special-q on the f_2-side. The first part of the relation collection will thus only concern the f_2-side. More precisely, the enumeration step and checking whether the norms are smaller than the sieve-cofactor only concern the norms N_2. However, once we have selected the sieve-survivors, it is necessary

to compute both norms N_1 and N_2 as they are the inputs to the batch algorithm.

Let us fix a special-q $\mathfrak{Q}$. The enumeration algorithm must be run for each prime ideal $\mathfrak{p}$ in K_2 up to $p_{\max} = 10,000,000$ and promising candidates, *i.e.*, promising $(a(\iota), b(\iota))$-pairs, correspond to vectors for which the norms have a small cofactor.

In order to keep track of hits, we use a sieving table indexed by the vectors $\mathbf{c}$ of the coefficients of $a(\iota), b(\iota)$ converted to an unsigned 64-bit integer value (in our code: `uint64_t pos = Vto64(c, R);`). Whenever $\mathbf{c}$ is in $\mathcal{L}_{\mathfrak{Q},\mathfrak{p}} \cap S_6(21)$ for a given $\mathfrak{p}$, we add an approximation of $\log p$ to the sieving table at the corresponding index, where p is the prime below $\mathfrak{p}$ (in our code: `sieve_table[pos] += (uint8_t)logp;`). We will then only look at promising candidates which are the $(a(\iota), b(\iota))$-pairs for which $\sum \log p > 50$ (in our code: `if (sieve_table[i] > 50)`).

For each of these promising candidates, we compute approximations of the corresponding norms, meaning using floating numbers to accelerate the computation, and finally output the $(a(\iota), b(\iota))$-pairs for which the cofactor $C_{\text{sieve}}(N_2(a(\iota) - b(\iota)\alpha_2)) \leq B_{\text{sieve}} = 60$ (in our code: `uint8_t cofac = logN - sieve_table[i] - sizeQ;`). Note that we also divide by the special-q. These are the sieve-survivors mentioned in Chapter 4. These sieve-survivors obtained at the end of enumeration as the output of `enum.c` are represented as

$$
\begin{array}{ccccc}
(\text{rel1}) & a_1(\iota) & b_1(\iota) & N_{1,f_2} & N_{1,f_1} \\
(\text{rel2}) & a_2(\iota) & b_2(\iota) & N_{2,f_2} & N_{2,f_1} \\
& \ldots
\end{array}
$$

where $a_i(\iota), b_i(\iota)$ are encoded into 64-bit integer values. The formatting is chosen to match the pre-existing format in CADO-NFS in order to directly branch ourselves into CADO-NFS's code for the batch and ECM algorithms.

Example 4 (Sieve-survivor). *We give below the example of a sieve-survivor outputted by* `enum.c`.

500147500015500123   499995499869499948   79504200998455080841334159240578251487503369685
62478165062664260822071748473

The polynomials corresponding to this sieve-survivor are

$$a(\iota) = 147 + 15\iota + 123\iota^2,$$

and

$$b(\iota) = -5 - 131\iota - 52\iota^2.$$

One can also see that the norm on the f_2-side is much larger than the norm on the f_1-side thus illustrating our choice to consider the special-q's on the f_2-side.

Concretely for our computation, we collected approximately 76,401 million sieve-survivors. Note that these survivors are dealt with on-the-fly in order to avoid storing them. In particular, they are removed just after the batch algorithm. Recall that at this stage of the algorithm, we also remove the K_h-unit duplicates and the ζ_2-duplicates.

The number of leaves and nodes enumerated.

We analyze our enumeration algorithm by computing the expected number of leaves and nodes for a fixed special-q as the value of p increases. As can be seen in Figure 5.1, the output of our enumeration algorithm matches the expected values given by the formulae in Chapter 4, Section 4.4.4. Both the amount of nodes and leaves decrease when p increases. The ratio between the amount of leaves and nodes also decreases with p as the probability of having a large prime factor in the norm gets smaller. We want this ratio to remain high as internal nodes correspond to (necessary) operations which do not produce any information as hits are seen only at the leaf level. We see that the estimation of the number of internal nodes is not precise. However, it gives a good idea of the general behavior of the algorithm. Furthermore the ratio of leaves per node indeed remains high, which is promising.

Number of leaves and nodes (left plot)

Number of leaves per node (right plot)
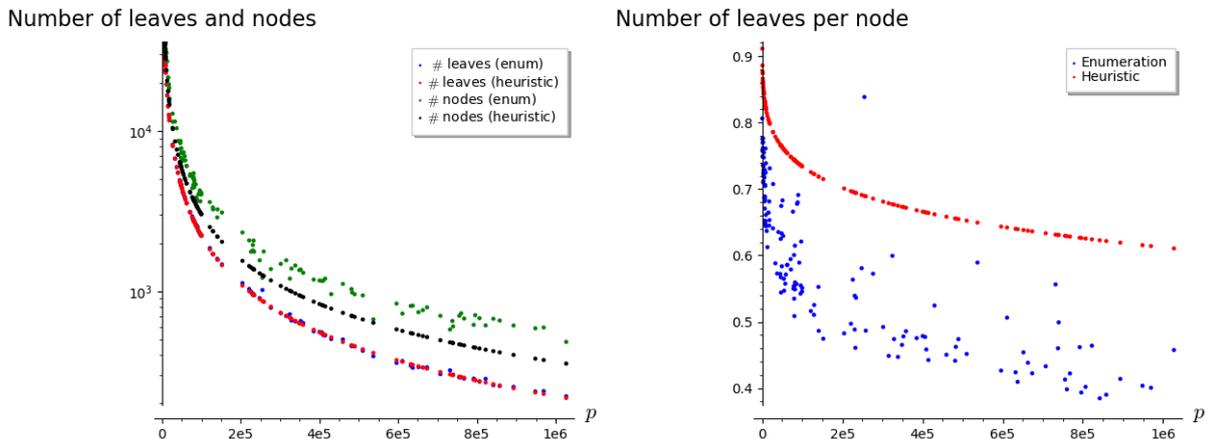
Figure 5.1: Number of leaves and nodes (left) and number of leaves per node (right) as a function of p for a fixed 24-bit special-q. We see that as p increases, both the number of nodes visited by the enumeration algorithm and the number of leaves decreases, as expected. We compare the output of our code with the formulae given in Chapter 4 using the Gaussian heuristic.

5.3.3 Balancing sieving, batch and ECM

Recall from Chapter 4, Figure 4.3, that the relation collection can be seen as a sequence of filters. The sieve-survivors outputted by the enumeration algorithm are now the inputs to the batch algorithm implemented in CADO-NFS. Running batch and ECM is done sequentially in CADO-NFS with the program `finishbatch`. The latter indeed has the option to use ECM on the batch-survivors or not. In our computation, we chose not to run ECM as the computation was efficient enough for the record to finish in reasonable time. Of course, further optimizing the parameters including the ECM algorithm is left for future work. We therefore select the batch-survivors with $p_{batch} = B = 2^{27}$ and $B_{batch} = 0$. Indeed, we want the batch-survivors to correspond to our relations and thus we want to ensure that all the norms have no cofactor left, *i.e.*, they completely factor into primes up to the smoothness bound $p_{batch} = B$. Finally, `finishbatch` will output the final relations encoded as follows

$$a(\iota), \quad b(\iota): \quad \sqcup, \sqcup, \sqcup, \cdots : \quad \sqcup, \sqcup, \sqcup, \cdots$$

where $\sqcup$ represents the prime factors of N_2 (before the :) and N_1 (after the :) expressed in hexadecimal.

Example 5 (Batch-survivor). *We give below the example of a batch-survivor outputted by* `finishbatch`. *This batch-survivor corresponds to the same $(a(\iota), b(\iota))$-pair as the sieve-survivor from Example 4.*

$$500147500015500123, 4999954998694999948:5,7,293,8cb,da3,15b5,19c7,3277,529d,10c33,6a1917,b3c06f,$$
$$5774ba9:1c9,75d9,be211,3c0dd21,5820169.$$

The total amount of relations outputted by the relation collection step as described in Algorithm 8 in Chapter 4 is a small percentage of the original sieve-survivors. Moreover, removing all the possible duplicates further reduces the final amount of relations. These numbers are described in Table 5.2.

	Sieve-survivors	Batch-survivors (removing K_h/ζ_2-dup.)	Removing special-q duplicates
# survivors	76 401M	18.69M	13.63M
% kept	0.013%	0.02 %	73 %

Table 5.2: Number of survivors after each step of the relation collection algorithm. The percentage is given with respect to the previous step. The percentage of sieve-survivors is taken with respect to $\mathrm{vol}(S_6(21)) \times \#$special-$q$'s.

This concludes the relation collection step which took 2.9 core-years or equivalently 25,300 core-hours. If we use the optimized code presented in Chapter 4, Algorithm 12, the relation collection takes only 23,300 core-hours, thus a gain of nearly 10 %.

5.3.4   From a set of relations to a matrix

The goal is now to transform the factorization of the norms into a factorization of ideals in order to construct the matrix for the linear system. Indeed, we are looking for the virtual logarithms of the elements of the factor basis, *i.e.*, ideals of small norms. The information we are looking for is given in Table 4.2 in Chapter 4 where we describe the matrix of relations.

Transforming the factorization of norms into a factorization in ideals is done with a pre-computed file `renumber`. The latter encodes all the ideals of the factor basis. One ideal $\mathfrak{p} = \langle p, \phi_h(\iota), x - \rho(\iota) \rangle$ corresponds to a line in the file encoded as follows.

$$p \quad 1 \text{ or } 2 \quad \deg \phi_h \quad \phi_{h_0} \quad \phi_{h_1} \quad \phi_{h_2} \quad \rho_0 \quad \rho_1 \quad \rho_2 \quad \texttt{index}$$

where p is a prime, 1 or 2 denotes the side we are considering (either the f_1-side or the f_2-side), the polynomial ϕ_h is expressed as

$$\phi_h(\iota) = \phi_{h_0} + \phi_{h_1}\iota + \cdots + \iota^{\deg \phi_h},$$

and corresponds to a monic irreducible factor of $h(\iota) \pmod{p}$. Recall that h is of degree 3 and thus $\deg \phi_h \leq 3$. The value $\rho(\iota)$ is a root of either f_1 or f_2 (depending on the side we are considering) in $\mathbb{F}_p[\iota]/\phi_h(\iota)$ and is expressed as

$$\rho(\iota) = \rho_0 + \rho_1\iota + \cdots + \rho_{\deg \phi_h - 1}\iota^{\deg \phi_h - 1}.$$

Finally, the last value `index` simply corresponds to an identifier of the ideal used afterwards to build the matrix (it will correspond to the column number of the ideal).

Remark 19. *In Chapter 4 we mentioned that the special-q ideals $\mathfrak{Q}$ and the ideals $\mathfrak{p}$ used for enumeration are of degree 1. However, when we look at batch-survivors, it is possible that some relations include ideals with $\deg \phi_h > 1$. The file* `renumber` *considers those ideals to keep as many potential relations as possible.*

This encoding matches the representation of the ideals given in Proposition 1 in Chapter 4 as we will now illustrate with two examples chosen from the `renumber` file used in our computation.

Example 6 (Ideal of degree 1). *Let us consider the following encoding of an ideal of degree 1.*

$$7 \quad 2 \quad 1 \quad 5 \quad 0 \quad 0 \quad 2 \quad 0 \quad 0 \quad 11$$

We are looking at an ideal $\mathfrak{p}$ over the prime 7 of degree 1. We have $\phi_h(\iota) = \phi_{h_0} + \iota$ and $\rho(\iota) = \rho_0 = 2$. Thus the ideal represented by this line is

$$\mathfrak{p} = (7, \iota + 5, x - 2).$$

It is affected to the 11th column in the matrix of relations.

Example 7 (Ideal of degree 2). *Let us consider the following encoding of an ideal of degree 2.*

$$7 \quad 1 \quad 2 \quad 3 \quad 2 \quad 0 \quad 5 \quad 3 \quad 0 \quad 15$$

We are looking at an ideal $\mathfrak{p}$ again over the prime 7 of degree 2. We have $\phi_h = 3 + 2\iota + \iota^2$ and $\rho(\iota) = 3\iota + 5$. Thus the ideal represented by this line is

$$\mathfrak{p} = (7, 3 + 2\iota + \iota^2, x - (3\iota + 5)).$$

The file `renumber` is thus a list of ideals encoded as explained above

$\mathfrak{p}_1$	p_1	1 or 2	$\deg\phi_{1,h}$	ϕ_{1,h_0}	ϕ_{1,h_1}	ϕ_{1,h_2}	$\rho_{1,0}$	$\rho_{1,1}$	$\rho_{1,2}$	index_1
$\mathfrak{p}_2$	p_2	1 or 2	$\deg\phi_{2,h}$	ϕ_{2,h_0}	ϕ_{2,h_1}	ϕ_{2,h_2}	$\rho_{2,0}$	$\rho_{2,1}$	$\rho_{2,2}$	index_2
$\vdots$										
$\mathfrak{p}_k$	$\approx 2^{27}$	1 or 2	$\deg\phi_{k,h}$	ϕ_{k,h_0}	ϕ_{k,h_1}	ϕ_{k,h_2}	$\rho_{k,0}$	$\rho_{k,1}$	$\rho_{k,2}$	index_k

for primes up to the smoothness bound $B = 2^{27}$.

A note on projective ideals. Let us consider the case where f_1 or f_2 is not unitary. Then if p is a prime that divides the leading coefficient of f_i, there exists a projective ideal above p in the factor basis. This is the case for f_2 and $p = 5$ in our computation. As in the classical NFS, this is handled in the code by considering the root 0 of $\tilde{f}_2$ where $\tilde{f}_2$ is the polynomial obtained by reversing the coefficients of f_2 and exchanging the roles of a and b.

This precomputed file is then used to convert the factorization of the norms into a factorization of ideals. Recall that the output of `finishbatch` which provides the factorization of the norms is of the form

$$(\text{rel1}) \quad a_1(\iota) \quad b_1(\iota): \quad p_{1,f_2}, p_{2,f_2}, \cdots : \quad p_{1,f_1}, p_{2,f_1}, \cdots$$
$$\cdots$$

where p_{i,f_2} (resp. p_{i,f_1}) are the primes in the factorization of N_2 (resp N_1). For each of these primes, we want to identify which ideal it corresponds to. The output of `renumber_tnfs` is thus a similar file where a relation is expressed as

$$(\text{rel1}) \quad a_1(\iota) \quad b_1(\iota): \quad \text{index}_1, \text{index}_2, \text{index}_3 \cdots$$
$$\cdots$$

where index_i corresponds to the index of the ideal corresponding to the prime factor p_{i,f_i}. We do not differentiate the indices that come from ideals from the f_1-side or the f_2-side.

Example 8 (Relation). *The batch-survivor from Example 5 finally outputs the relation*

5001477500015500123,499995499869499948:0,4,c,272,73f,a94,fee,1274,1d04,2369,465f,e8
30d,17b4da,a21070,1c2,2a53,1f5f1,71c9b1,a33bba,

where the terms after : correspond to the indices of the prime ideals. If the multiplicity is greater than 1, the index is written multiple times.

Remark 20. *Every relation starts with the index 0. The latter corresponds to the ideal $J_1 J_2$ where J_i is the smallest ideal such that for all $(a(\iota), b(\iota))$-pair, the ideal $J_i(a(\iota) - b(\iota)\alpha_i)$ is an integral ideal. As in the classical NFS, simply adding this column filled with ones in the matrix is enough to accommodate the fact that f_1 or f_2 might not be monic.*

This outputted file gives us part of the matrix used in the linear algebra step. The latter can be seen as the following table

Relations	$\mathfrak{p}_1$	$\mathfrak{p}_2$	$\mathfrak{p}_3$	$\cdots$
rel1	1	0	0	$\cdots$

where index_i allows to place 1 in the corresponding column when the ideal appears in the factorization. The missing part of the input matrix to the linear algebra step are the columns that correspond to Schirokauer maps which we discuss in Section 5.4.2. It remains to answer the following question.

Question 67. *How do we translate a prime p in the factorization of the norm into its corresponding ideal over p?*

This is done by checking whether an ideal $\mathfrak{p}$ divides $\phi(\iota, \alpha_i) = a(\iota) - b(\iota)\alpha_i$. Indeed, Proposition 1 from Chapter 4 tells us that the unique ideal of $\mathcal{O}_i$ above p which divides $\phi(\iota, \alpha_i)$ is the ideal $(p, \alpha_i - r(\iota))$ with $r(\iota) \equiv \frac{a(\iota)}{b(\iota)}$ (mod p). More precisely, we have

$$\mathfrak{p} | (a(\iota) - b(\iota)\alpha_i) \iff \frac{a(\iota)}{b(\iota)} \equiv \rho(\iota) \pmod{(p, \phi_h(\iota))}. \tag{5.1}$$

Recall that we have the following structure of tower of number fields.

$$
\begin{array}{ccccc}
\mathbb{Q} & \longrightarrow & K_h & \longrightarrow & K_i \\
p & & (p, \phi_h(\iota)) & & \mathfrak{p} = (p, \phi_h(\iota), x - \rho(\iota))
\end{array}
$$

> **Question 68.** *What does Equation 5.1 mean?*

For ideals of degree 1. In this case, we know that $\phi_h(\iota)$ is of degree 1 since $\mathfrak{p}$ is. This also means that $\rho(\iota) = \rho \in \mathbb{Z}/p\mathbb{Z}$. Equation 5.1 can thus be re-written as

$$\mathfrak{p} | (a(\iota) - b(\iota)\alpha_i) \iff \frac{a(r_h)}{b(r_h)} \equiv \rho \pmod{p},$$

where r_h is a root of $\phi_h(\iota)$, *i.e.*, we have $r_h = -\phi_{h_0} \pmod{p}$. Hence, in order to test for divisibility by the ideal $\mathfrak{p}$, we simply verify

$$a(r_h) - \rho b(r_h) \equiv 0 \pmod{p}.$$

For ideals of degree 2. This case is slightly more complex as $\phi_h(\iota)$ is now also of degree 2. If we have

$$a(\iota) \equiv \rho(\iota)b(\iota) \pmod{p}, \pmod{\phi_h(\iota)},$$

then Equation 5.1 is satisfied. Note that the above equation is equivalent to

$$(a(\iota) \pmod{\phi_h(\iota)}) \equiv (\rho(\iota)b(\iota) \pmod{\phi_h(\iota)}) \pmod{p}.$$

Both sides of this equivalence are polynomials in ι of degree less than $\deg \phi_h(\iota)$. Thus it suffices to check whether the coefficients of these two polynomials are equal modulo p. This narrows down to checking two equalities of elements modulo p.

Remark 21. *For ideals of degree 3, the same reasoning than above applies and results in testing three equations modulo p.*

Checking the divisibility by ideals of the factor basis is done with `renumer_tnfs.cpp`. Concretely we can summarize this process as follows.

For each prime factor $p_j^{g_j}$ in the factorization of N_i, we first need to find all ideals $\mathfrak{p}$ over $p_j^{g_j}$ that divides $\phi(\iota, \alpha_i)$. To do so, for a given p_j we proceed as follows.

1. use `renumber` file to find all ideals over p_j, meaning lines in `renumber` that start with the prime p_j.

2. check whether the ideal divides $\phi(\iota, \alpha_i)$, as explained above.

3. form a list $L = \{$ideals over p_j that divides $\phi(\iota, \alpha_i)\}$

Next, we need to make sure that the multiplicities coincide. Let us first illustrate this with a simple example.

Example 9. *Suppose that in the factorization of the norm N_i, we have a prime factor p^4, i.e, $p^4 | N_i \in \mathbb{Z}$. Now suppose we have found the ideals $\mathfrak{p}_1, \mathfrak{p}_2$ and $\mathfrak{p}_3$ above p that divide $a(\iota) - b(\iota)\alpha_i$, i.e., $L = \{\mathfrak{p}_1, \mathfrak{p}_2, \mathfrak{p}_3\}$. Consider the following two cases.*

Case 1. *The degrees of the ideals $\mathfrak{p}_i$ are $\deg \mathfrak{p}_1 = 1, \deg \mathfrak{p}_2 = 1$ and $\deg \mathfrak{p}_3 = 2$. Then $\sum \deg \mathfrak{p}_i = 4$, and thus the ideal factorization corresponding to p^4 is $\mathfrak{p}_1 \mathfrak{p}_2 \mathfrak{p}_3$.*

Case 2. *Now consider the case where the degrees are $\deg \mathfrak{p}_1 = 1, \deg \mathfrak{p}_2 = 1$ and $\deg \mathfrak{p}_3 = 1$. Then $\sum \deg \mathfrak{p}_i = 3$, and thus the ideal factorization corresponding to p^4 is $\mathfrak{p}_1^{e_1} \mathfrak{p}_2^{e_2} \mathfrak{p}_3^{e_3}$ where $e_i = 1$ for two ideals and $e_i = 2$ for one of them.*

When $\sum_{\mathfrak{p} \in L} \deg \mathfrak{p} \neq g_j$ as in Case 2, we unfortunately do not know how to identify the exponents e_i of the ideals. Thus we throw away the relation. More specifically, we have the following steps to deal with multiplicities.

1. if $|L| = 1$, output the index of the unique ideal.

2. if $|L| > 1$: output indices of ideals in L if and only if $\sum \deg \mathfrak{p}_j = g_j$.

Remark 22. *Fortunately, the amount of relations we throw away because we are not able to produce the ideal factorization only corresponds to a loss of 2% in our computation. The number of batch-survivors given in Table 5.2 is 18.69M, and removing the relations for which we are not able to obtain the ideal factorization we finally have 18.25M batch-survivors.*

We are left with two questions that our code does not consider.

Question 69. *How can we find the multiplicities of the ideals in the factorization and thus keep those relations? And what happens when $p | disc(f_i)$?*

The answer to these two questions is not hard in theory and efficient algorithms are given in Cohen's textbook [Coh12]. However, since the loss of relations is acceptable, we preferred to keep our code simple at this stage of the development.

Now that we have all the indices of the ideals present in the factorizations of the norms N_i for many polynomials ϕ, we can move to the next step: linear algebra. Following CADO-NFS's tradition, we deal with the Schirokauer maps after filtering but it could be done before.

5.4 Linear algebra

Before starting the linear algebra step, the matrix must undergo a few modifications in order to speed up the resolution of the system. This is done by a step called filtering. More precisely, the aim of filtering is to reduce the size of the matrix of relations without modifying its kernel.

5.4.1 Duplicates and filtering

Dealing with special-q duplicates. As mentioned in Chapter 4, only ζ_2-duplicates and K_h-unit duplicates can be dealt with prior to constructing the matrix. To remove special-q duplicates, we simply compare the ideal factorization of each relation and remove identical lines (after :) in the file containing all of our relations. Before eliminating special-q duplicates we had 18.25M batch-survivors. Removing these duplicates decreased the amount of survivors to 13.63M, as seen in Table 5.2, which corresponds to a loss of 27%.

Filtering. The matrix of relations is now ready to be sent to CADO-NFS's filter. We have 15.21M ideals in the factor basis, and thus the input matrix to CADO-NFS's filter is a matrix of size $13.63M \times 15.21M$. Note that not all the ideals will intervene in the relations. Indeed, in our computations, only 13.18M ideals were involved. The input matrix to the filtering step is illustrated in Figure 5.2.
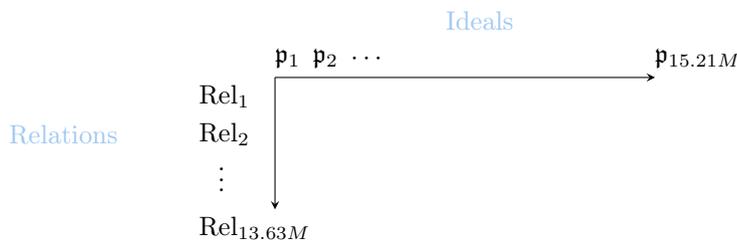
Figure 5.2: Input matrix to filtering step.

The goal of filtering is to both reduce the size of the matrix and make it square. Filtering in CADO-NFS comprises of two steps: purge and merge.

Purge. The first part of filtering consists in removing columns that only contain zero coefficients. Indeed as mentioned above, only 87% of the ideals of the factor basis appear in relations. The rest will lead to zero-columns that are deleted.

Moreover, the purge step removes columns (and corresponding lines) that contain a unique element. These columns correspond to prime ideals that occur only once in all the relations. The latter are named *singletons*. This does not remove any information as we now explain.

> **Question 70.** *How will we compute the virtual logarithm of singletons?*

Because it is the unique ideal in a relation for which we do not know the virtual logarithm, the latter will be computed after the linear algebra step from the other virtual logarithms involved. It is thus sufficient to keep track of the relations removed by the filter in order to recompute the missing virtual logarithms before the descent phase.

The second part of purge consists in transforming the matrix into a nearly square matrix. In general, as can be seen in our computation, once we remove the unnecessary ideals, the number of lines (corresponding to the number of relations) is greater than the number of columns (*i.e.,* the number of ideals involved in the relations).

In order to remove the additional lines, the filter proceeds with the so-called clique removal algorithm associated with a weight function that determines which clique is more advantageous to remove. The term *clique* is wrongly associated to the similar notion from graph theory which denotes a connected component of a graph. In the context of filtering, a clique usually refers to a set of relations (*i.e.,* lines) such that removing one of these lines creates as many singletons as the size of the clique. Removing cliques is done until the matrix is (nearly) square. Details are given in [Bou15].

In our computation, we started with 13.63M lines in the matrix. Once we removed the singletons, we were left with only 5.21M lines. Hence, purge reduced the number of lines in our matrix by approximately 62%. Thus we see that even if the purge step (and more generally filtering) is not present in the complexity analysis of TNFS, it is of significant importance in practice for the feasibility of the linear algebra step.

> **Question 71.** *Why do we want a square matrix?*

The block-Wiedemann algorithm which we use for the linear algebra step (and other iterative algorithms in general) takes as input a square matrix. Indeed, the algorithm considers powers of the input matrix which requires a square matrix. The filter must thus produce a nearly square matrix. The *nearly* comes from the fact that we will have to add columns for the Schirokauer maps. Since we know in advance how many columns we will add (7 in our computation), we keep as many additional lines.

Merge. The merge step corresponds to a structured Gaussian elimination. It aims at further reducing the matrix size by performing linear combinations of the rows of the input matrix. Indeed, combining lines of the matrix can create singletons that are then removed. This does not change the squareness of the matrix as both a column and a line are removed for each singleton. The resulting matrix is smaller but also denser due to possible added non-zero coefficients. A density parameter is usually chosen to determine the average number of coefficients per line. This serves as a stopping condition for the elimination process.

In our computation, the input matrix to the merge process is a square matrix of dimension 5.21M as mentioned above. After Gaussian elimination up to a density of 100 coefficients per line, the size of the matrix is decreased to 1.73M and if we eliminate up to 150 coefficients per line, the size is even further decreased to 1.51M.

> **Question 72.** *How do we choose the density parameter?*

Recall that the complexity of the linear algebra step with Wiedemann's algorithm is $O(N^2\lambda)$ where λ is the average number of non-zero coefficients per line. In theory, it is thus possible to optimize λ in order to minimize the overall cost of the linear algebra step. In practice, this parameter does not affect the

complexity much and $\lambda = 150$ is chosen in our computation based on previous observations.

After filtering, we are left with 1.51M relations and a $(1.51M + 7) \times 1.51M$ dimension matrix. The entire filtering step removed 89% of the relations (or 92% if we count before the removal of the duplicates).

Remark 23. *Because the filter in CADO-NFS was originally implemented for factorization, it only works for a matrix to which we have not yet added the columns corresponding to the Schirokauer maps. Hence, it is important to keep in mind the operations done during the filter to the matrix in order to later report them to the columns corresponding to the Schirokauer maps.*

5.4.2 Schirokauer maps

Recall from Chapter 4, Section 4.2.3 the notion of Schirokauer maps. The latter allow us to include in the matrix the missing information necessary to compute the virtual logarithms of our ideals. There are as many Schirokauer maps as the rank of units in K_i. For our computation, this means 2 on the f_1-side and 5 on the f_2-side.

Moreover, we explained in Chapter 4, Section 4.2.3 that Schirokauer maps in the context of TNFS were simply built by constructing an isomorphism from K_i to a number field K_{F_i} where F_i is of degree $\deg h \times \deg f_i$. Consider an isomorphism

$$\Phi_i : \quad \begin{matrix} K_i & \to & K_{F_i} \\ a(\iota) - b(\iota)x & \mapsto & \Phi_i(a(\iota) - b(\iota)x) \end{matrix}$$

with

$$\Phi_i(a(\iota) - b(\iota)x) = a_0\Phi_i(1) + a_1\Phi_i(\iota) + a_2\Phi_i(\iota^2) - b_0\Phi_i(x) - b_1\Phi_i(x\iota) - b_2\Phi_i(x\iota^2).$$

The elements $\Phi_i(y) \in K_{F_i}$ are polynomials in x of degree 12 for $i = 1$ and 6 for $i = 2$. The program `prepareMapData.mag` creates $6 \times \deg F_i$ matrices with the coefficients of $\Phi_i(1), \Phi_i(\iota), \Phi_i(\iota^2), \Phi_i(x), \Phi_i(x\iota)$ and $\Phi_i(x\iota^2)$. Hence, computing the isomorphisms Φ_i that maps sieving polynomials ϕ to the number field K_{F_i} narrows down to taking linear combinations of the pre-computed elements of the $6 \times \deg F_i$ matrix.

> **Question 73.** *How are the polynomials F_i and the matrices defined?*

The program `prepareMapData.mag` starts by computing K_{F_i}. The function `AbsoluteField($K_i$)` in Magma returns a number field isomorphic to K_i defined as an absolute extension (over $\mathbb{Q}$). The defining polynomial of K_{F_i} is thus F_i. For example, we have $K_1 = K_h/(x^4 + 1)$ and

$$K_{F_1} = \mathbb{Q}[X]/(x^{12} - 4x^{10} + 4x^9 + 9x^8 - 12x^7 + 10x^6 - 36x^5 + 4x^3 + 44x^2 + 4x + 1).$$

Once the number fields K_{F_i} are defined, the evaluations of Φ at $1, \iota, x, \cdots$, are simply given by Magma's representation of these elements in K_{F_i}. For example $\Phi_i(\iota)$ is given by $K_{F_i}!K_h$.`1`.

Remark 24. *The values $\Phi_i(1), \Phi_i(\iota), \cdots$ have denominators. Since Schirokauer maps, as implemented in CADO-NFS, only require integers, the program* **prepareMapData.mag** *directly multiplies the coefficients by the least common multiplier of these denominators.*

This denominator-clearing can be made completely transparent by choosing Schirokauer maps Λ_i that evaluate to zero over $\mathbb{Q}$. This corresponds to the *legacy mode* in CADO-NFS. We will discuss in Section 5.7 the possibility to choose even further specific Schirokauer maps that could lead to faster linear algebra.

> **Open Question 3.** *Is there an optimal way to choose Schirokauer maps in the context of TNFS?*

Computing the Schirokauer maps, *i.e.*, filling up the seven columns, took 40 core-hours by parallelizing the process over 64 virtual cores.

5.4.3 Solving the system

In Chapter 4, we described Wiedemann's algorithm that efficiently solves sparse systems of linear equations. In practice, its block-variant is used to parallelize the process. Recall that one of the most costly steps of Wiedemann's algorithm is constructing a Krylov's sequence $\{\mathbf{a}^\intercal M^i \mathbf{b}\}_{i=1}^{2N}$, where M is the input matrix of dimension N. Fortunately, this construction can be parallelized. In the block variant, the vectors $\mathbf{a}$ and $\mathbf{b}$ are replaced by matrices and the sequence computed is thus $\{AM^iB\}$ for $i = 1, 2, \cdots$ where A, B are matrices of smaller dimensions, more precisely, we have $A \in \mathbb{Z}/\ell\mathbb{Z}^{N\times m}$ and $B \in \mathbb{Z}/\ell\mathbb{Z}^{N\times n}$ for given parameters $n, m > 0$ known as blocking factors.

Then n separate nodes will compute the sequence $\{AM^ib_j\}_{i=1}^{L}$, where $L = \lceil N/m\rceil + \lceil N/n\rceil + \lceil m/n + n/m\rceil$. The total cost of the algorithm remains the same, meaning $O(\lambda N^2)$, however distributed over n nodes. The main difference resides in the fact that we now have a sequence $\{AM^iB\}_{i=1}^{L}$ of matrices whereas we had coefficients in Wiedemann's algorithm. In 2002, Thomé [Tho02] proposed a variant of Berlekamp-Massey algorithm to compute linear generators for sequences of matrices. His algorithm runs in sub-quadratic time. Hence, similarly as in Wiedemann's original algorithm, we find a linear generator that plays the same role as the minimal polynomial of M, from which we can reconstruct a vector of the kernel.

The input matrix to the block-Wiedemann algorithm is the concatenation of the output of the filter step and the columns from the Schirokauer map to which we applied the same linear transformations as in the filter. The resulting matrix is square. The matrix outputted by the filter after both purge and merge is of size 1.51M. From the Schirokauer maps, we know that 7 columns are added. This is illustrated in Figure 5.3.
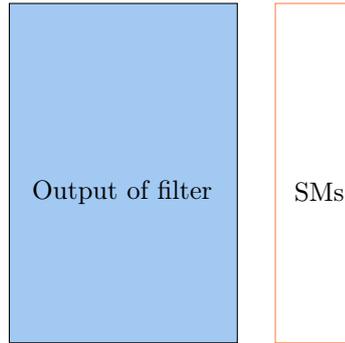
Figure 5.3: Input to the block-Wiedemann algorithm

Question 74. *Is this matrix really sparse?*

The main issue with the input matrix given in Figure 5.3 is the fact that the columns coming from the Schirokauer maps are far from sparse. However, as was shown in [JP16], it is possible to solve the linear system without changing the asymptotic complexity of the block-Wiedemann algorithm. Indeed, the main idea is to only apply block-Wiedemann's algorithm to the sparse part of the matrix, thus ignoring the Schirokauer columns. Additionally, one should consider the matrix B in the sequence $\{AM^iB\}_{i=1}^{L}$ as the matrix formed by these Schirokauer columns. Hence, the value n is chosen to be exactly the number of Schirokauer maps, in our case 7. Further details are given in [JP16].

The computation of each sequence $\{AM^ib_j\}$ on a single node took about 19,440 seconds, *i.e.,* 5.4 hours and thus the total time to compute the sequence is $19,440 \times 7 \times 32$ seconds which corresponds to 1,210 core hours (where $n = 7$ is the number of Schirokaueur maps and 32 is the number of CPU cores per node). The reconstruction of the linear generator took 2,400 seconds, *i.e.,* 40 minutes and finally the reconstruction of the solution took 19,380 seconds, *i.e.,* 5.4 hours. Overall, the linear algebra step thus took $(19,440 \times 7 + 2,400 + 19,380) \times 32 = 5,051,520$ core seconds or equivalently 1,403 core hours.

One can see that the most expensive steps of the linear algebra part are the computation of the sequences and the solution step where the sparse matrix-vector multiplication is the most costly operation.

We now have the virtual logarithms of the elements of the factor basis in a file `log_reconstructed.txt`. The encoding of the ideals are the same as in the file `renumber` explained previously, and the last coefficient correspond to the virtual logarithm of the ideal.

Example 10. *For example, the following line can be found in* `log_reconstructed.txt`.

> 5   1   1   2   0   0   0   0   0   3   17371686314656041575273159549226841957739008784327397

The ideal $\mathfrak{p} = \langle 5, \iota + 2, x \rangle$ *in the factor basis of* `index` *3 has virtual logarithm*

$$\text{vlog } \mathfrak{p} = 17371686314656041575273159549226841957739008784327397.$$

5.5   Descent step and discrete logarithm of the target

Now that we have solved our system, we are ready for the descent step. Recall that

$$\ell = 182429353442218325399060814128481197043091244242217403$$

is the modulus with which we work corresponding to the prime order of a subgroup of the multiplicative group of our target finite field.

> **Question 75.** *What is our target element?*

Recall that we chose as target in $\mathbb{F}_{p^6}$ an element whose decimals are taken from π, that is

$$
\begin{aligned}
\texttt{target} =&(31415926535897932384626433 + 83279502884197169399375105\iota \\
&+ 82097494459230781640628620\iota^2) + x(89986280348253421170679821 \\
&+ 48086513282306647093844609\iota + 55058223172535940812848111\iota^2).
\end{aligned}
$$

> **Question 76.** *How did we choose the generator g?*

The representation of $\mathbb{F}_{p^6}$ is naturally obtained as a degree-2 extension (defined by $f_2(x)$) of the field $\mathbb{F}_{p^3}$ defined by $h(\iota)$, meaning $\mathbb{F}_{p^6} \cong \mathbb{Z}[\iota][x]/(I)$, where I is a common irreducible factor of f_1 and f_2. We took as a generator the element $g = x + \iota$. This element $g = x + \iota$, lifted in the field defined by f_1 is a unit (of infinite order). This allowed us to easily compute its virtual logarithm as it can be found using the virtual logarithms outputted by the linear algebra step and an additional Schirokauer map computation.

Remark 25. *Note that the virtual logarithm of the generator is computed in an arbitrary basis, say γ, the same basis for which we have the virtual logarithms of the elements of the factor basis outputted by the linear algbebra step.*

In order to get the discrete logarithm of our target element in base g, we first compute the discrete logarithm of the target in base γ. Because we know the discrete logarithm of g in base γ we can finally obtain the discrete logarithm of our target in base g.

As mentioned in Chapter 4, we use Guillevic's algorithm [Gui19] to optimize the initial splitting step. The descent starts by a smoothing step which required 45 core hours to generate 64M candidates and 10 core hours to identify an element $s \in \mathbb{F}_{p^6}^*$ such that its lift to K_1 has a 35-bit smooth norm.

The factors of s greater than 27-bit for which we did not have the virtual logarithms yet were descended in a single special-q step. This descent was done using `enum.c` and `finishbatch` with a larger radius $R = 33$. Because of the small amount of factors concerned, the time was negligible. Thus in total, the descent step took 55 core hours. The overall time in core hours of the computation is reported in Table 5.3.

We finally find the discrete logarithm of our target element:

$$\log(\texttt{target}) = 76272808168753222977667479701383785303538529763155498.$$

Relation Collection	Linear algebra	Schirokauer maps	Descent	Overall time
23,300	1,403	40	55	24,798

Table 5.3: Overall time of our record computation in core hours.

In order to confirm the validity of our computation, we verify that $g^{\log(\texttt{target})} = \texttt{target}$ is indeed true, modulo ℓ^{th} powers, since we computed the discrete logarithm only modulo ℓ. The verification script for SageMath is given below.

SageMath verification script to check the discrete log of Tower NFS computation in $\mathrm{GF}(p^6)$.

```
p = 135066410865995223349603927
ell = 1824293534422183253990608141284811970430912442421740 3
cof = (p**6-1) // ell

# Construction of the finite field:
Fp = GF(p)
FpX.<X> = Fp[]
h = X**3 - X + 1
Fp3.<ι> = Fp.extension(h)
Fp3.<Y> = Fp3[]
phiY = Y**2 + 64417723306991464419622353*Y + 1
Fp6.<x> = Fp3.extension(phiY)

# Generator and target element:
gen = ι + x
target = ( 31415926535897932384626433 + 83279502884197169399375105*ι
+ 82097494459230781640628620*ι**2) + x*( 89986280348253421170679821 +
48086513282306647093844609*ι + 55058223172535940812848111*ι**2)

# Logarithm of target, as computed with TNFS:
log_target = 7627280816875322297766747970138378530353852976315498

# Check that this is indeed the log modulo ell:
(gen**cof)**log_target == (target**cof)
```

5.6 Comparing with NFS computations

An important aspect for DLP (and factoring) records is the size of the norms involved in the computation. Indeed, recall that the complexity of the algorithm is highly dependent on the smoothness probability of the norms. If norms are smaller, then the relation collection step will be more efficient as it will be faster to find enough B-smooth norms.

5.6.1 Size of norms in our TNFS computation

One significant advantage of TNFS over NFS is the smaller size of the norms considered. We report in Figure 5.4 the average bitsize of the norms N_1 and N_2/q for the relations we obtained at the end of the relation collection step. Note that because we used the special-q method on the f_2-side, which has larger norms, we also divide by the size of the special-q.

On average, the norms N_1 are around 95 bits and the norms N_2/q are around 154 bit. The overall size of the product of the norms is thus around 249 bits.
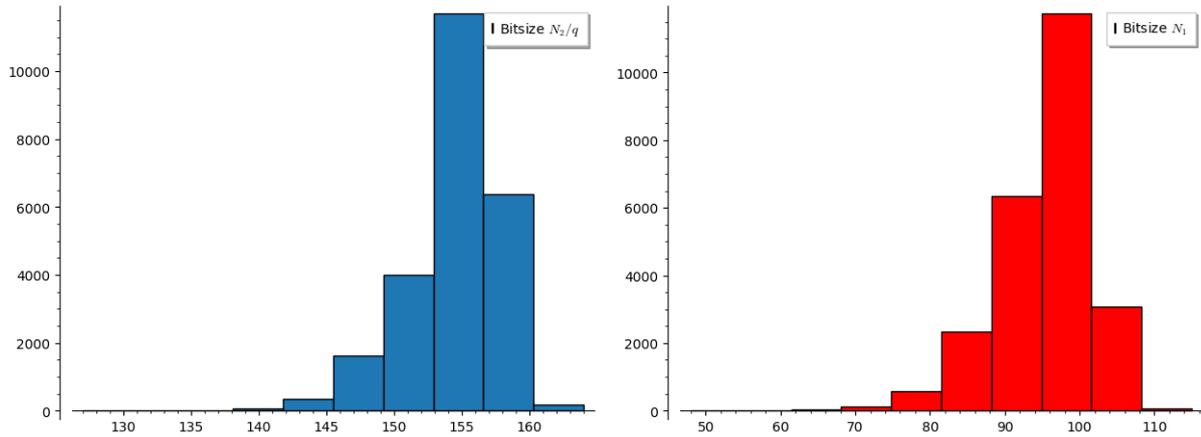
Figure 5.4: The bitsize of the norms coming from the relations in our computation. On the left, the norms N_2/q have average bitsize 154 bits and on the right the norms N_1 have average bitsize 95 bits.

Our record computation takes place in a 521-bit finite field. We would like to consider the following questions.

> **Question 77.** *How is our TNFS computation comparable with factoring an integer of the same size? How does it compare to using NFS for DLP in a 521-bit prime field?*

The git repository of CADO-NFS provides optimized parameters for both factoring and discrete logarithms computations for various target-sizes. We will use these parameters for our next comparisons.

5.6.2 Comparing with factoring with NFS

We look at the parameters from the file params.c155 and the polynomials from c155.poly in CADO-NFS. This means we are considering an integer N with 155 digits, which corresponds to a 512-bit integer, hence slightly smaller than the size we want. A 521-bit integer would have around 157 digits, hence we are close enough for the purpose of this comparison.

Polynomial selection from c155.poly.

$$f_1(x) = 745920x^5 - 2076894693938x^4 - 681801484930531955x^3 + 1614628025120092091914179x^2$$
$$+ 1889048721679082659393958 18184x - 587869192028594861338213432986476 00,$$

and

$$f_2(x) = 7756938953438894 2609247x - 54797380596259623814 1689365703.$$

Sieving parameters from params.c155. The $p_{\max}$ on both sides is taken to be a 25-bit integer, the smoothness bound B is taken to be 2^{29}, and the cofactors are around 62 bits. The value $I = 2^{14}$ gives the sieving space $A = [-I/2, I/2] \times [0, I/2]$, whereas we have $A = \text{vol}(S_6(R))$ for TNFS. To have the same volume, we would need to pick $R = 17$. Recall that our computation is done with $R = 21$ and thus norms with TNFS and $R = 17$ would be slightly smaller than those reported above. Finally, $q_{\min} = 15,470,309$.

What we do. We randomly sample vectors $c = (i, j)$ such that $|i| < I/2$ and $0 < j < J$. Here, we take $I = J = 2^{14}$ and estimate the bitsize of the norms. To do so, we define the lattice L_q for a given special-q (we choose $q = 15,470,327$, which corresponds to the first prime after the given $q_{\min}$) as follows

$$L_q = [(q, 0), (r, s)]$$

where r is a root of $f_1 \mod q$ and s is the skewness factor, here equal to 574,720. The skewness factor s in the lattice is an additional parameter that comes into play only in factorization computations. Indeed, in this case the coefficients of the polynomial f_1 are disbalanced as can be seen above. The coefficients of high degree are smaller than those of small degree. This effect is compensated by adapting the sieving

area and taking a larger than b in an (a, b)-pair, see [BBKZ16]. We then reduce the lattice to have smaller coefficients,

$$M_q = LLL(L_q) = LLL([(q, 0), (r, s)]).$$

For a given vector $c = (i, j) \in S$, we compute $v = c \times M_q = (a, b)$ and define $\phi = a - bx$. We then compute the norms $N_1 = f_1(a/b)b^5/q$ as we consider the special-q on the f_1-side and $N_2 = f_2(a/b)b^2$. The bitsize of the norms N_1, N_2 are on average

$$\log N_1/q \approx 167,$$

and

$$\log N_2 \approx 113.$$

Thus, the product of the norms is around 280 bits which is larger than the one considered in our computation for which we had 249 bits.

5.6.3 Comparing with DLP with NFS

We would like to make the same comparison with a DLP computation done with NFS.

Polynomial selection. We use Joux-Lercier to obtain a polynomial f_1 of degree 4 and a polynomial f_2 of degree 3. We have

$$f_1(x) = 6x^4 + x^3 - 8x^2 - 3x + 6,$$

and

$$\begin{aligned} f_2(x) = & -640712643068849916118590009153886700616x^3 - 1158843791903746768523484547831308883186x^2 \\ & + 38282308072029980108426725373486173 9469x - 1452949298428843681969169925389 5818591. \end{aligned}$$

Sieving parameters. We choose the parameters from CADO-NFS `parameters/dlp/params.p155` (line 53). The p_{max} value is taken to be 23 bits (resp. 26 bits) on the f_1-side (resp. on the f_2-side). The smoothness bound is 2^{29} (resp. 2^{30}) on the f_1-side (resp. on the f_2-side). The cofactor size is 81 bits (resp. 64 bits) on the f_1-size (resp. on the f_2-side). Finally, the value $I = 2^{14}$ gives the sieving space $A = [-I/2, I/2] \times [0, I/2]$, similarly as for factoring.

What we do. We randomly sample vectors $c = (i, j)$ such that $|i| < I/2$ and $0 < j < J$. Again, we take $I = J = 2^{14}$. We define the lattice L_q for a given special-q (we choose $q = 561, 907$, no values were given in the parameters) as follows

$$L_q = [(q, 0), (r, 1)],$$

where r is a root of $f_2 \bmod q$. We reduce it to have smaller coefficients

$$M_q = LLL(L_q) = LLL([(q, 0), (r, 1)]).$$

For any vector $c \in S$, we compute $v = c \times M_q = (a, b)$ and define $\phi = a - bx$. We then compute the norms $N_1 = \text{Res}(\phi, f_1) = f_1(a/b)b^4$ and $N_2 = \text{Res}(\phi, f_2) = f_2(a/b)b^3/q$. The bitsize of the norms N_1, N_2 are thus

$$\log N_1 \approx 95,$$

and

$$\log N_2/q \approx 175.$$

on average. Thus the product of the norms is 270 bits, slightly less than for factoring but still more than with TNFS, as expected, where we had 249. Recall that to consider an equal volume, we would have chosen a smaller R and thus the bitsize of the norms would be even smaller.

Hence we can see that TNFS indeed considers smaller norms for nearly-equivalent computations for both DLP and factoring, which is a significant motivation to consider this algorithm for further record computations. This fact was already known to hold asymptotically, but we confirm this is already the case (and very much so!) for sizes amenable to practical computation.

5.6.4 Comparing with other high-dimension sieves

In Chapter 4, we compared our enumeration procedure with the 3-dimensional sieving algorithms of Laurent Grémy [Gré18] and McGuire and Robinson [MR21]. In particular, we argued that considering a d-sphere instead of a d-orthotope would lead to a faster sieving step. In Table 5.4 we compare our computation with theirs, keeping in mind that we also do not use the same algorithm. Our computation is much faster despite the larger size of the finite field considered.

Parameters	[GGMT17]	[MR21]	This work
Algorithm	NFS	NFS	TNFS
Field size (bits)	422	423	521
Sieving dimension	3	3	6
Sieving time	201,600	69,120	**23,300**

Table 5.4: Comparison of the relation collection step in core hours with [GGMT17] and [MR21] for $\mathbb{F}_{p^6}$.

5.7 Conclusion

This chapter presented the first implementation and computation of a discrete logarithm using TNFS. This record corresponds to the larger characteristic p for which a discrete logarithm computation is performed in $\mathbb{F}_{p^6}$. The latest record in $\mathbb{F}_{p^6}$ was given in [MR21] in 2020 for a 423-bit finite field. We mention here a few directions for future work.

Using Galois action. A factor 2 in the sieving time can easily be gained by considering Galois actions. Indeed, consider the special-q $\mathfrak{q} = (q, r_h, r_{f_2})$ and $(a(\iota), b(\iota))$ a pair leading to a relation for $\mathfrak{q}$. Then, the special-q $\mathfrak{q}^\sigma = (q, r_h, r_{f_2}^{-1})$ will also lead to the relation given from the pair $(b(\iota), a(\iota))$ where it suffices to invert the roles of a and b.

In previous record computations such as in [BGGM15], the use of the Galois action on the ideals also saved a factor 4 in the linear algebra step. However, it is not clear how this can be translated to the TNFS setup. A better understanding of the interaction between Schirokauer maps and the Galois interaction seems to be a necessary step towards this improvement.

Road to bigger sizes. One main advantage of TNFS over NFS when computing discrete logarithms is the much smaller bitsize of the norms involved as was illustrated above. This indicates that for a comparable computational cost, it would be possible to compute discrete logarithms in much larger finite fields using TNFS rather than NFS.

> **Open Question 4.** *Up to which size of finite field extension is it reasonable to believe TNFS performs efficiently?*

The limitations of our algorithm will quickly arise when the dimension of the lattice increases. Indeed, whereas enumerating in small dimension lattices is very fast, enumeration is done $\#$ special-$q \times p_{\max}$ times approximately and thus a slight increase in the time of a single enumeration process impacts by quite a lot the overall time of relation collection. This is also true if $p_{\max}$ is increased for example, or if we require more special-q's. However, considering dimensions up to 8 seems reachable at this point. A significant amount of work remains to study the interactions between the many parameters of all the sub-algorithms of TNFS in order to optimize it and aim for larger finite fields.

Part III

Partial key recovery from side-channel information

Chapter 6

Overview of partial key recovery methods

You are dangling in a rope sling hung from the ceiling of a datacenter in an undisclosed location, high above the laser tripwires criscrossing the floor. You hold an antenna over the target's computer, watching the bits of their private key appear one by one on your smartwatch display. Suddenly you hear a scuffling at the door, the soft beep of keypad presses. You'd better get out of there! You pull your emergency release cable and retreat back to the safety of the ventilation duct. Drat! You didn't have time to get all the bits! Mr. Bond is going to be very disappointed in you. Whatever are you going to do?

Side-channel attacks targeting cryptography may leak only partial or indirect information about the secret keys of the protocols. There are a variety of techniques in the literature for recovering secret keys from partial information. In this chapter, we survey several of the main families of partial key recovery algorithms for RSA, (EC)DSA, and (elliptic curve) Diffie-Hellman, the public-key cryptosystems in common use today. We categorize the known techniques by the structure of the information that is learned by the attacker, and give simplified examples for each technique to illustrate the underlying ideas. This chapter serves as an introduction to two side-channel attacks presented in Chapters 7 and 8.

The chapter is joint work with Nadia Heninger and is available on Eprint, report 2020/1506.

Contents

6.1 Introduction

In a side-channel attack, an attacker exploits side effects from computation or storage to reveal ostensibly secret information. Many side-channel attacks stem from the fact that a computer is a physical object in the real world, and thus computations can take different amount of time [Koc96], cause changing power consumption [KJJ99], generate electromagnetic radiation [QS01], or produce sound [GST14], light [FH08], or temperature [HS14] fluctuations. The specific character of the information that is leaked depends on the high- and low-level implementation details of the algorithm and often the computer hardware itself: branch conditions, error conditions, memory cache eviction behavior, or the specifics of capacitor discharges.

The first work on side-channel attacks in the published literature did not directly target cryptography [EL85], but since Kocher's work on timing and power analysis in the 90s [Koc96, KJJ99], cryptography has become a popular target for side-channel work. However, it is rare that an attacker will be able to simply read a full cryptographic secret through a side channel. The information revealed by many side-channel attacks is often indirect or incomplete, or may contain errors.

Thus in order to fully understand the nature of a given vulnerability, the side-channel analyst often needs to make use of additional cryptanalytic techniques. The main goal for the cryptanalyst in this situation is typically to wonder the following.

> **Question 78.** *I have obtained the following type of incomplete information about the secret key. Does it allow me to efficiently recover the rest of the key?*

Unfortunately there is not a one-size-fits-all answer: it depends on the specific algorithm used, and on the nature of the information that has been recovered.

The goal of this chapter is to collect some of the most useful techniques in this area together in one place, and provide a reasonably comprehensive classification on what is known to be efficient for the most commonly encountered scenarios in practice. That is, this chapter is a non-exhaustive survey and a concrete tutorial with motivational examples. Many of the algorithmic papers in this area give constructions in full generality, which can sometimes obscure the reader's intuition about why a method works. Here, we aim to give minimal working examples to illustrate each algorithm for simple but nontrivial cases. We restrict our focus to public-key cryptography, and in particular, the algorithms that are currently in wide use and thus the most popular targets for attack: RSA, (EC)DSA, and (elliptic curve) Diffie-Hellman.

Throughout this work, we will illustrate the information known for key values as follows:

<div align="center">

Known bits

Most significant bits   Least significant bits

</div>

The organization of this survey is given in Table 6.1.

6.1.1 Motivation

While this chapter is mostly operating at a higher level of mathematical abstraction than the side-channel attacks that we are motivated by, we will give a few examples of how attackers can learn partial information about secrets.

Modular exponentiation. All of the public-key cryptographic algorithms we discuss involve modular exponentiation or elliptic curve scalar addition operating on secret values. For RSA signatures, the victim computes $s = m^d \pmod{N}$ where d is the secret exponent. For DSA signatures, the victim computes a per-signature secret value k and computes the value $r = g^k \pmod{p}$, where g and p are public parameters. For Diffie-Hellman key exchange, the victim generates a secret exponent a and computes the public key exchange value $A = g^a \pmod{p}$, where g and p are public parameters.

Scheme	Secret information	Bits known	Technique	Section
RSA	$p \geq 50\%$ most significant bits		Coppersmith's method	§6.2.2.2
RSA	$p \geq 50\%$ least significant bits		Coppersmith's method	§6.2.2.3
RSA	p middle bits		Multivariate Coppersmith	§6.2.2.4
RSA	p multiple chunks of bits		Multivariate Coppersmith	§6.2.2.4
RSA	$> \log \log N$ chunks of p		Open problem	
RSA	$d \pmod{p-1}$ MSBs		Coppersmith's method	§6.2.2.7
RSA	$d \pmod{p-1}$ LSBs		Coppersmith's method	§6.2.2.7 and §6.2.2.3
RSA	$d \pmod{p-1}$ middle bits		Multivariate Coppersmith	§6.2.2.7 and §6.2.2.4
RSA	$d \pmod{p-1}$ chunks of bits		Multivariate Coppersmith	§6.2.2.7 and §6.2.2.4
RSA	d most significant bits		Not possible	§6.2.2.8
RSA	$d \geq 25\%$ least significant bits		Coppersmith's method	§6.2.2.9
RSA	$\geq 50\%$ random bits of p and q		Branch and prune	§6.2.3.1
RSA	$\geq 50\%$ of bits of $d \pmod{p-1}$ and $d \pmod{q-1}$		Branch and prune	§6.2.3.2
(EC)DSA	MSBs of signature nonces		Hidden Number Problem	§6.3.2
(EC)DSA	LSBs of signature nonces		Hidden Number Problem	§6.3.2
(EC)DSA	Middle bits of signature nonces		Hidden Number Problem	§6.3.2
(EC)DSA	Chunks of bits of signature nonces		Extended HNP	§6.3.2.4
EC(DSA)	Many bits of nonce		Scales poorly	
Diffie-Hellman	Most significant bits of shared secret g^{ab}		Hidden Number Problem	§6.4.2
Diffie-Hellman	Secret exponent a		Pollard kangaroo method	§6.4.3
Diffie-Hellman	Chunks of bits of secret exponent		Open problem	

Table 6.1: Visual table of contents for key recovery methods for public-key cryptosystems.

Naive modular exponentiation algorithms like Square-and-Multiply operate bit by bit over the bits of the exponent: each iteration will execute a square operation, and if that bit of the exponent is a 1, will execute a multiply operation. We recall the pseudo-code for the left-to-right Square-and-Multiply algorithm in Algorithm 13.

Algorithm 13 Left-to-right Square-and-Multiply algorithm

Input: $x, k, n \in \mathbb{N}$, with $x < n$ and $k = (k_{i-1}, k_{i-2}, \cdots, k_0)$ its bit representation
Output: $x^k \pmod{n}$

1: $c \leftarrow 1$
2: **for** $j = i - 1$ to 0 **do**
3:      $c \leftarrow c^2 \pmod{n}$ ▷ Square-
4:      **if** $k_i = 1$ **then**
5:          $c \leftarrow c \times x$ ▷ and-Multiply
6: **return** c

There exists a right-to-left version of Square-and-Multiply for which the bits are read from k_0 to k_{i-1}. The overall structure of the algorithm is very similar with simple adjustments. If the bit is 1, then we multiply (mod n) the value c by a value y which is initialized at x, and then we square y mod n. If the bit is 0, we only square y.

More sophisticated modular exponentiation algorithms precompute a digit representation of the exponent using non-adjacent form (NAF), windowed non-adjacent form (wNAF) [Möl03], sliding windows, or Booth recoding [Boo51] and then operate on the precomputed digit representation. We now recall the NAF representation and its windowed variant wNAF as well as how it is used for exponentiation as we will use it in Chapter 8. We refer to [Gor98] for more details.

The (w)NAF representation. The execution time of the textbook Square-and-Multiply algorithm depends on the number of non-zero bits in the exponent since the multiplication operation only happens when the bit is 1. One way of reducing the execution time in then to represent the exponent using another representation which reduces the number of non-zero digits and thus the number of multiplications.

The NAF representation precisely aims at reducing the Hamming weight of a scalar, *i.e.,* the number of non-zero digits in the representation of that scalar.

> **Definition 26** (NAF representation)**.** *For any $k \in \mathbb{Z}$, a representation $k = \sum_{j=0}^{\infty} k_j 2^j$ is called a NAF if $k_j \in \{0, \pm 1\}$ and $k_j k_{j+1} = 0$ for all $j \geq 0$.*

Every k has a unique NAF representation. The algorithm to convert k into its NAF form is presented in Algorithm 14.

The windowed-NAF (wNAF) representation even further reduces the Hamming weight of the scalar by considering larger digits. The digits k_j from the definition given above are not taken from the set $\{0, \pm 1\}$ anymore but from the larger set $\{0, \pm 1, \pm 3, \cdots, \pm 2^w - 1\}$ where w is the chosen window. The scalar k is converted into wNAF form using Algorithm 15.

Algorithm 14 NAF algorithm	**Algorithm 15** wNAF representation
Input: $k \in \mathbb{Z}^+$	**Input:** $k \in \mathbb{Z}^+$, $w \in \mathbb{N}$
Output: NAF representation of k	**Output:** wNAF representation of k
1: $i = 0$	1: $i = 0$
2: **while** $k > 0$ **do**	2: **while** $k > 0$ **do**
3: **if** $k \pmod 2 = 1$ **then**	3: **if** $k \pmod 2 = 1$ **then**
4: $k_i = 2 - (k \pmod 4)$	4: $k_i = k \pmod{2^{w+1}}$
5: $k = k - k_i$	5: **if** $k_i \geq 2^w$ **then**
6: **else**	6: $k_i = k_i - 2^{w+1}$
7: $k_i = 0$	7: $k = k - k_i$
8: $k = k/2$	8: **else**
9: $i = i + 1$	9: $k_i = 0$
	10: $k = k/2$
	11: $i = i + 1$

We give an example of the integer 23 expressed in binary, NAF and 3NAF representations.

Example 11. *In binary, we can write*

$$23 = 2^4 + 2^2 + 2^1 + 2^0 = (1, 0, 1, 1, 1),$$

whereas in NAF-representation, we have

$$23 = 2^5 - 2^3 - 2^0 = (1, 0, -1, 0, 0, -1).$$

With $w = 3$, the wNAF representation gives

$$23 = 2^4 + 7 \times 2^0 = (1, 0, 0, 0, 7).$$

The number of non-zero coefficients in these three representations of 23 is minimized for the 3NAF representation.

The (w)NAF representation can then be used to further improve the execution time of exponentiation. We provide the pseudo-code for the exponentiation algorithm using the (w)NAF form in Algorithm 16.

Algorithm 16 Exponentiation with wNAF form

Input: window size w and integer $k = (k_0, k_1, \cdots, k_n)$ in its wNAF form, $x \in \mathbb{N}$.
Output: x^k
1: Pre-compute $x_d = x^d$ for all $d = 3, 5, 7, \cdots 2^{w-1} - 1$.
2: $c \leftarrow 1$.
3: **for** $i = 0$ to n **do**
4:    $c = c^2$
5:    **if** $k_i \neq 0$ **then**
6:        $c \leftarrow c \times x_{k_i}$
7: **return** c

The (w)NAF representation is well suited to fast exponentiation and thus has been used in many implementations. Further improvements to these methods are given in [Möl03].

Booth recoding. The Booth recoding is a similar recoding technique for a scalar k where recoding with a window size w represents the scalar as a sequence of digits k_i such that $-2^{w-1} \leq k_i \leq 2^{w-1}$ and $k = \sum_i 2^{wi} k_i$. This encoding will be used for the scalar multiplication in the quoting enclave of SGX, see

Chapter 7.

We now go over microarchitectural side-channel attacks with a particular emphasize on cache timing attacks.

Microarchitectural side-channel attacks. Microarchitectural side-channel attacks recover secret information by artificially creating observable contentions between different CPU execution units. Introduced over a decade ago [Pag02, TTMH02, TSS$^+$03, Per05, Ber05, OST06], they have since been used to break the security of many real-world systems. Examples of targets include cryptographic primitives [ASK07, AS08, BvSY14], measurement of keystroke timings [LGS$^+$16, LGS$^+$17], website fingerprinting [GZES17]. Researchers have also demonstrated attacks within a variety of ecosystems: from within the target's browser [OKSK15, AKM$^+$15, GMM16], from inside or against SGX enclaves [XCP15, LSG$^+$17, VBWK$^+$17, SWG$^+$17, MIE17, BMD$^+$17, MES18], or even on third party compute clouds [LYG$^+$15, IAES15, IGI$^+$16]. More recently, microarchitectural cache attacks have been combined with speculative execution to read sensitive data across security domains, such as kernel data [LSG$^+$18, KGG$^+$18].

In order for such an attack to work, there must be microarchitectural elements (eg; cache lines, hardware buffers) shared between the victim and the attacker's application, and the elements must further have data-dependent state, *i.e.*, the latter changes depending on the data being processed. The most commonly observed microarchitectural elements are shared cache components. Cache attacks exploit contention on a shared cache to infer secret information from unsuspecting processes. They can be realized in certain scenarios wherein adversary and benign threads share access to the same cache. Moreover, the aforementioned data-dependent state must be observable via one (or multiple) side channels.

Many microarchitectural side-channel attacks use variations in execution time as the source of leakage. This type of attack has been extensively researched since Kocher's [Koc96] seminal work. Timing attacks rely on the influence that microarchitectural elements have on execution time of certain operations. For example, a malicious process can, by measuring access time to shared or its own data, infer if a victim process has accessed related regions of the cache. The resulting side channel leaks information about the memory access patterns of the victims which can be exploited to attack implementations of cryptographic schemes such as AES [OST06], RSA [AS08], and (EC)DSA [BvSY14].

Public-key cryptographic protocols are a common target for timing-attack research. For a timing attack to successfully recover the secret key, the running time of certain cryptographic operations must depend to some extent on the secret key. Moreover, key recovery often relies on the assumption that a large amount of the same operation can be carried out without the key being changed.

Cache attacks on modular exponentiation. There are many variants of these attacks, but they all share in common that the attacker is able to execute code on a CPU that is co-located with the victim process and shares a CPU cache. While the victim code executes, the attacker measures the amount of time that it takes to load information from locations in the cache, and thus deduces information about the data that the victim process loaded into those cache locations during execution. In the context of the modular exponentiation or scalar addition algorithms discussed above, a cache attack on a vulnerable implementation might reveal whether a multiply operation was executed at a particular bit location if the attacker can detect whether the code to execute the multiply instruction was loaded into the cache. Alternatively, for a pre-computed digit representation of the number, the attacker may be able to use a cache attack to observe the digit values that were accessed [ASK07, AS08, BvSY14].

We now recall three main types of cache attacks.

Evict+Time The cache attack methodology measures the difference in execution time of the victim's code to determine whether the victim accessed a memory location that maps to one of the evicted cache sets. Evict+Time has been successfully used to attack cryptographic protocols such as AES [OST06, TOS10], for example. The methodology is as follows:

1. **Evict.** The attacker evicts a specific cache set of the victim from the cache.

2. **Time.** A variation in the execution time indicates that the victim is accessing data which maps to the cache set previously evicted by the attacker.

Prime+Probe This cache attack methodology measures the difference in time it takes to refill a given cache set. It does not rely on any shared memory addresses between the attacker and the victim [Per05, OST06]. Apart from cryptographic protocols, Prime+Probe has also been demonstrated to be effective at obtaining sensitive information in cloud environments [LYG+15, IAES15, YKSA15]. The Prime+Probe methodology works as follows:

1. **Prime.** The attacker fills relevant cache sets by sequentially loading memory addresses that map to the same set.

2. **Victim Memory Access.** The attacker waits for the victim to perform secret-dependent stores to memory. When the victim stores this data, it evicts some of the attacker's cache lines from the targeted sets.

3. **Probe.** The attacker reloads the previously cached memory addresses and measures the access times to each cache set. A longer access time to a set corresponds to a victim access to that particular set. When the value of a secret variable affects the access patterns to memory, these patterns reveal information about the secret.

Flush+Reload This attack [YF14, YB14] follows an attack methodology similar to Prime+Probe, however it does not require the attacker to evict any cache set or line. Instead, it measures the difference in time when reloading a memory line. The attacker targets a specific region in memory which is shared with the victim. This attack does not require the knowledge of physical addresses, which is a common limitation of other cache attacks. Flush+Reload has been also extensively used to attack cryptographic primitives such as RSA [CAPGATB19], DSA [GBY16], ECDSA [ABF+16, GB17], AES [GBK11], and PRNGs [CKP+19]. The methodology goes as follows.

1. **Flush.** The monitored memory line is flushed from the cache hierarchy.

2. **Victim Memory Access.** The attacker waits for the victim to access the memory line.

3. **Probe.** The spy reloads the memory line, measuring the time it takes. If the reloading operation is fast, then the victim has accessed the memory line and thus the line is available in the cache. On the other hand, a long reload means that the victim has not accessed the memory line, and thus the line has to be brought from memory.

Flush+Flush [GMWM16] and Evict+Reload [GSM15, LGS+16] are two variants of Flush+Reload which we do not detail in this chapter.

Other attacks on modular exponentiation. Other families of side channels that have been used to exploit vulnerable modular exponentiation implementations include power analysis and differential power analysis attacks [KJJ99, KJJR11], electromagnetic radiation [QS01], acoustic emanations [GST14], raw timing [Koc96], photonic emission [FH08], and temperature [HS14]. These attacks similarly exploit code or circuits whose execution varies based on secrets.

Cold boot and memory attacks. An entirely different class of side-channel attacks that can reveal partial information against keys include attacks that may leak the contents of memory. These include cold boot attacks [HSH+08], DMA (Direct Memory Access), Heartbleed, and Spectre/Meltdown [LSG+18, KHF+19]. While these attacks may reveal incomplete information, and thus serve as theoretical motivation for some of the algorithms we discuss, most of the vulnerabilities in this family of attacks can simply be used to read arbitrary memory with near-perfect precision, and cryptanalytic algorithms are rarely necessary.

Length-dependent operations. A final vulnerability class is implementations whose behavior depends on the length of a secret value, and thus variations in the behavior may leak information about the number of leading zeros in a secret. A simple example is copying a secret key to a buffer in such a way that it reveals the bit length of a secret key. In another example, the Raccoon attack observes that TLS versions 1.2 and below strips leading zeros from the Diffie-Hellman shared secret before applying the key derivation function, resulting in a timing difference depending on the number of hash input blocks required for the length of the secret. [MBA+20]

6.2 Key recovery methods for RSA

Let us first focus on the well-known RSA cryptosystem and start by recalling how it works.

6.2.1 RSA Preliminaries

Parameter Generation. To generate an RSA key pair, implementations typically start by choosing the public exponent e. By far the most common choice is to simply fix $e = 65537$. Some implementations use small primes like 3 or 17. Almost no implementations use public exponents larger than 32 bits. This means that attacks that involve brute forcing values less than e are generally feasible in practice.

In the next step, the implementation generates two random primes p and q such that $p-1$ and $q-1$ are relatively prime to e. The public modulus is $N = pq$. The private exponent is then computed as

$$d = e^{-1} \pmod{(p-1)(q-1)}.$$

The public key is the pair (e, N). In theory, the secret key is the pair (d, N), but in practice many implementations store keys in a data structure including much more information. For example, the PKCS#1 private key format includes the fields p, q, $d_p = d \pmod{p-1}$, $d_q = d \pmod{q-1}$, and $q_{inv} = q^{-1} \pmod{p}$ to speed encryption using the Chinese Remainder Theorem.

Encryption and Signatures. In textbook RSA, Alice encrypts the message m to Bob by computing $c = m^e \pmod{N}$. In practice, the message m is not a "raw" message, but has first been transformed from the content using a padding scheme. The most common encryption padding scheme in network protocols is PKCS#1v1.5, but OAEP [BR95] is also sometimes used or specified in protocols. To decrypt the encrypted ciphertext, Bob computes $m = c^d \pmod{N}$ and verifies that m has the correct padding.

To generate a digital signature, Bob first hashes and pads the message he wishes to sign using a padding scheme like PKCS#1v1.5 signature padding (most common) or PSS (less common). Let m be the hashed and padded message of this form. Then Bob generates the signature as $s = m^d \pmod{N}$. Alice can verify the signature by computing the value $m' = s^e \pmod{N}$ and verifying that m' is the correct hashed and padded value.

Since encryption and signature verification only use the public key, decryption and signature generation are the operations typically targeted by side-channel attacks.

RSA-CRT. To speed up decryption, instead of computing $c^d \pmod{N}$ directly, implementations often use the Chinese remainder theorem (CRT). RSA-CRT splits the exponent d into two parts $d_p = d \pmod{p-1}$ and $d_q = d \pmod{q-1}$.

To decrypt using the Chinese remainder theorem, Alice would compute $m_p = c^{d_p} \pmod{p}$ and $m_q = c^{d_q} \pmod{q}$. The message can be recovered with the help of the pre-computed value $q_{inv} = q^{-1} \pmod{p}$ by computing

$$m = m_p q q_p + m_q (1 - q q_p) = (m_p - m_q) q q_{inv} + m_q \pmod{N}.$$

This is called Garner's formula [Gar59].

> **Question 79.** *What are the relationships between the different RSA key elements?*

For the purpose of secret key recovery, we typically assume that the attacker knows the public key. RSA keys have a lot of mathematical structure that can be used to relate the different components of the public and private keys together for key recovery algorithms. The RSA public and private keys are related to each other as

$$ed \equiv 1 \pmod{(p-1)(q-1)}.$$

The modular equivalence can be removed by introducing a new variable k to obtain an integer relation

$$ed = 1 + k(p-1)(q-1) = 1 + k(N - (p+q) + 1).$$

We know that $d < (p-1)(q-1)$, so $k < e$. The value of k is not known to the attacker, but since generally $e \leq 65537$ in practice it is efficient to brute force over all possible values of k.

For attacks against the CRT coefficients d_p and d_q, we can obtain similar relations

$$ed_p = 1 + k_p(p-1) \quad \text{and} \quad ed_q = 1 + k_q(q-1), \tag{6.1}$$

for some integers $k_p < e$ and $k_q < e$. Brute forcing over two independent 16-bit values can be burdensome, but we can relate k_p and k_q as follows.

Rearranging the two relations, we obtain $ed_p - 1 - k_p = k_p p$ and $ed_q - 1 - k_q = k_q q$. Multiplying these together, we get

$$(ed_p - 1 + k_p)(ed_q - 1 - k_q) = k_p k_q N.$$

Reducing the above modulo e, we get

$$(k_p - 1)(k_q - 1) \equiv k_p k_q N \pmod{e}. \tag{6.2}$$

Thus given a value for k_p, we can solve for the unique value of $k_q \pmod{e}$, and for applications that require brute forcing values of k_p and k_q we only need to brute force at most e pairs [IGI+15].

The multiplier k also has a nice relationship to these values. Multiplying the relations from Equation 6.1 together, we have

$$(ed_p - 1)(ed_q - 1) = k_p(p-1)k_q(q-1)$$

Substituting $(p-1)(q-1) = (ed-1)/k$ and reducing modulo e, we can relate the coefficients as

$$k \equiv -k_p k_q \pmod{e}.$$

Any of the secret values p, q, d, d_p, d_q, or q_{inv} suffices to compute all of the other values when the public key (N, e) is known.

From either p or q, computing the other values is straightforward. For small e, N can be factored from d by computing

$$ed = 1 + k(p-1)(q-1) = 1 + k(N - (p+q) + 1) \tag{6.3}$$

The integer multiplier k can be recovered by rounding $\lceil (ed-1)/N \rceil$. Once k is known, then Equation 6.3 can be rearranged to solve for $s = p + q$. Once s is known, we have

$$(p+q)^2 = s^2 = p^2 + 2N + q^2,$$

and

$$s^2 - 4N = p^2 - 2N + q^2 = (p-q)^2.$$

Then N can be factored by computing $\gcd((p+q) - (p-q), N)$.

When e is small, p can be computed from d_p as

$$p = \gcd((ed_p - 1)/k_p + 1, N)$$

where k_p can be brute forced from 1 to e. If k_p is not known and is too large to brute force, with high probability for a random a,

$$p = \gcd(a^{ed_p - 1} - 1, N).$$

Factoring from q_{inv} is more complex. As noted in [HS09], q_{inv} satisfies $q_{inv}q^2 - q \equiv 0 \pmod{N}$, and q can be recovered using Coppersmith's method, described below.

6.2.2 RSA Key Recovery with Consecutive bits known

This section covers techniques for recovering RSA private keys when large contiguous portions of the secret keys are known. The main technique used in this case is lattice basis reduction introduced in Chapter 2.

For the key recovery problems in this section, we can typically recover a large unknown chunk of bits of an unknown secret key value p, $d \pmod{p-1}$, or d. We typically assume that the attacker has access to the public key (N, e) but does not have any other auxiliary information, about q or $d \pmod{q-1}$, for example.

Knowledge of large contiguous portions of secret keys is unlikely to arise in side channels that involve noisy measurements, but could arise in scenarios where secrets are being read out of memory that got corrupted in an identifiable region. They can also help make attacks more efficient if a high cost is paid to recover known bits.

6.2.2.1 Warm-up: Lattice attacks on low-exponent RSA with bad padding.

The main algorithmic technique used for RSA key recovery with contiguous bits is to formulate the problem as finding a small root of a polynomial modulo an integer, and then to use lattice basis reduction to solve this problem.

In order to introduce the main tool of using lattice basis reduction to find roots of polynomials, we will start with an illustrative example for the concrete application of breaking small-exponent RSA with known padding. In later sections we will show how to modify the technique to cover different RSA key recovery scenarios.

The original formulation of this problem is due to Coppersmith [Cop96a]. Howgrave-Graham [HG97] gave a dual approach that we find easier to explain and easier to implement. May's survey [May10] contains a detailed description of both Coppersmith and Howgrave-Graham algorithms.

To set up the problem, we have an integer N, and a polynomial $f(x)$ of degree k that has a root r modulo N, that is, $f(r) \equiv 0 \pmod{N}$. We wish to find r. Finding roots of polynomials can be done efficiently modulo primes [LLL82], so this problem is easy to solve if N is prime or the prime factorization of N is known. The Coppersmith/Howgrave-Graham methods are generally of interest when the prime factorization of N is not known. It gives an efficient algorithm for finding all *small* roots (if they exist) modulo N of unknown factorization.

Problem setup.   For our toy example, we will use the 96-bit RSA modulus

$$N = \texttt{0x98664cf0c9f8bbe76791440d}$$

and $e = 3$. Consider a broken PKCS#1v1.5-esque RSA encryption padding scheme that pads a message m as

$$\text{pad}(m) = \texttt{0x01FFFFFFFFFFFFFFFFFF00} \ || \quad m$$

with the message m concatenated at the end. Now imagine that we have obtained a ciphertext

$$c = \texttt{0xeb9a3955a7b18d27adbf3a1}$$

and we wish to recover the unknown message m.
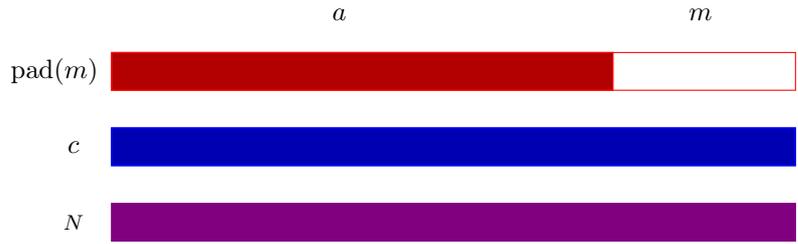
Figure 6.1: Illustration of low-exponent RSA message recovery attack setup. The attacker knows the public modulus N, a ciphertext c, and the padding a prepended to the unknown message m before encryption. The attacker wishes to recover m.

Cast the problem as finding roots of a polynomial. Let

$$a = \texttt{0x01FFFFFFFFFFFFFFFF0000}$$

be the known padding string, offset to the correct byte location. We also know the length of the message, in this case $m < 2^{16}$. Thus we have that $c = (a+m)^3 \pmod{N}$, for unknown small m. Let $f(x) = (a+x)^3 - c$. We have set up the problem so that we wish to find a small root m satisfying $f(m) \equiv 0 \pmod{N}$ for the polynomial

$$f(x) = x^3 + \texttt{0x5fffffffffffffffffd0000}x^2 + \texttt{0x6f1c485f406ba1c069460efe}x$$
$$+ \texttt{0x203211880cdc43afe1c5c5f9}$$

We have reduced the coefficients modulo N so that they will fit on the page.

Construct a lattice. Let the coefficients of f be $f(x) = x^3 + f_2x^2 + f_1x + f_0$. Let $M = 2^{16}$ be an upper bound on the size of the root m. We construct the matrix

$$B = \begin{pmatrix} M^3 & f_2M^2 & f_1M & f_0 \\ 0 & NM^2 & 0 & 0 \\ 0 & 0 & NM & 0 \\ 0 & 0 & 0 & N \end{pmatrix}$$

We then apply the LLL lattice basis reduction algorithm to the matrix. The shortest vector of the reduced basis is

$$\mathbf{v} = (-\texttt{0x66543dd72697}M^3, -\texttt{0x35c39ac91a11c04}M^2, \texttt{0x3f86f973d67d25eae138}M,$$
$$-\texttt{0x10609161b131fd102bc2a8})$$

Extract a polynomial from the lattice and find its roots. We then construct the polynomial g using the coefficients of the vector $\mathbf{v}$,

$$g(x) = -\texttt{0x66543dd72697}x^3 - \texttt{0x35c39ac91a11c04}x^2$$
$$+ \texttt{0x3f86f973d67d25eae138}x - \texttt{0x10609161b131fd102bc2a8}$$

The polynomial g has one integer root, $\texttt{0x42}$, which is the desired solution for m.

This specific 4×4 lattice construction works to find roots up to size $N^{1/6}$ as we will explain just below. For the small key size we used in our example, this is only 16 bits, but since it scales directly with the modulus size, this same lattice construction would suffice to learn 170 unknown bits of message for a 1024-bit RSA modulus, or 341 bits of message for a 2048-bit RSA modulus. Lattice reduction on a 4×4 lattice basis is instantaneous.

The rows of this matrix correspond to the coefficient vectors of the polynomials $f(x)$, Nx^2, Nx, and N. We know that each of these polynomials evaluated at $x = m$ will be 0 modulo N. Each column is scaled by a power of M, so that the ℓ_1 norm of any vector in this lattice is an upper bound on the value of the corresponding (un-scaled) polynomial evaluated at m. For a vector $\mathbf{v} = (v_3 M^3, v_2 M^2, v_1 M, v_0)$ in the lattice,

$$|f(m)| = |v_3 m^3 + v_2 m^2 + v_1 m + v_0| \leq |v_3 M^3| + |v_2 M^2| + |v_1 M| + |v_0| = ||\mathbf{v}||_1$$

for any $|m| \leq M$.

We have constructed the lattice so that every polynomial g we extract from it has the property that $g(m) \equiv 0 \pmod{N}$. We have also constructed our lattice so that the length of the shortest vector in a reduced basis will be less than N. This imposes a condition on the bound M as we explain just below. The only integer multiple of N less than N is 0, so by construction the polynomial corresponding to this short vector satisfies $g(m) = 0$ over the integers, not just modulo N. Since finding roots of polynomials over the integers, rationals, reals, and complex numbers can be done in polynomial time, we can compute the roots of this polynomial and check which of them is our desired solution.

This method will always work if the lattice is constructed properly. That is, we need to ensure that the reduced basis will contain a vector of length less than N. For this example, $\det B = M^6 N^3$. Heuristically, the LLL algorithm will find a vector of ℓ_2 norm $||\mathbf{v}||_2 \leq 1.02^n (\det B)^{1/\dim B}$. We ignore the 1.02^n factor, and the difference between the ℓ_2 and ℓ_1 norms for the moment. Then the condition we wish to satisfy is

$$g(m) \leq ||\mathbf{v}||_2 \leq (\det B)^{1/n} < M$$

For our example, we have $(\det B)^{1/\dim L} = (M^6 N^3)^{1/4} < N$. Solving for M, this will be satisfied when $M < N^{1/6}$. In this case, N has 96 bits, and m is 16 bits, so the condition is satisfied.

This can be extended to $N^{1/e}$, where e is the degree of the polynomial f by using a larger dimension lattice. Howgrave-Graham's dissertation [HG98] and May's survey [May10] give detailed explanations of this method and improvements.

6.2.2.2 Factorization from consecutive bits of p.

In this section we show how to use lattices to factor the RSA modulus N if a large portion of contiguous bits of one of the factors (without loss of generality p) is known.
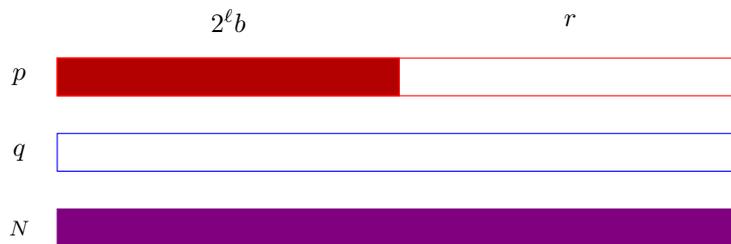
Figure 6.2: Factorization of $N = pq$ given contiguous known most significant bits of p.

Coppersmith solves this problem in [Cop96a] but we find the reformulation from Howgrave-Graham as "approximate integer common divisors" [HG01] simpler to apply, and will give that construction here.

Problem setup. Let $N = pq$ be an RSA modulus with equal-sized p and q. Choosing an example with numbers small enough to fit on the page, we have a 240-bit RSA modulus

$$N = \texttt{0x4d14933399708b4a5276373cb5b756f312f023c43d60b323ba24cee670f5.}$$

We assume N is known and we also know a large contiguous portion of the most significant bits b of p, so that $p = a + r$, where we do not know r but do know the value $a = 2^\ell b$. Here $\ell = 30$ is the number of unknown bits, or equivalently the left shift of the known bits.

In our example, we have

$$a = \texttt{0x68323401cb3a10959e7bfdc0000000}.$$

Cast the problem as finding the roots of a polynomial. Let $f(x) = a + x$. We know that there is some value r such that $f(r) = p \equiv 0 \pmod{p}$. We do not know p, but we know that p divides N and we know N. We also know that the unknown r is small, and in particular $|r| < R$ for some bound R that is known. Here, $R = 2^{30}$.

Construct a lattice. We can form the lattice basis

$$B = \begin{pmatrix} R^2 & Ra & 0 \\ 0 & R & a \\ 0 & 0 & N \end{pmatrix}.$$

We then run the LLL algorithm on our lattice basis B. Let $\mathbf{v} = (v_2 R^2, v_1 R, v_0)$ be the shortest vector in the reduced basis. In our example, we get the vector

$$\begin{aligned} \mathbf{v} =& (-\texttt{0x17213d8bc94}R^2, -\texttt{0x1d861360160a4f86181}R, \\ & \texttt{0xf9decdc1447c3f3843819a5d}). \end{aligned}$$

Extract a polynomial and find the roots. We form a polynomial $f(x) = v_2 x^2 + v_1 x + v_0$. For our example,

$$\begin{aligned} f(x) =& -\texttt{0x17213d8bc94}x^2 - \texttt{0x1d861360160a4f86181}x \\ & + \texttt{0xf9decdc1447c3f3843819a5d}. \end{aligned}$$

We can then calculate the roots of f. In this example, f has one integer root, $r = \texttt{0x873209}$. We can then reconstruct $a + r$ and verify that $\gcd(a + r, N)$ factors N.

This 3×3 lattice construction works for any $|r| < p^{1/3}$, and directly scales as p increases. In our example, we chose p and q so that they have 120 bits, and r has 30 bits. However, this same construction will work to recover 170 bits from a 512-bit factor of a 1024-bit RSA modulus, or 341 bits from a 1024-bit factor of a 2048-bit RSA modulus.

Question 82. *Why does this work?*

The rows of this matrix correspond to the coefficient vectors of the polynomials $x(x + a)$, $x + a$, and N. We know that each of these polynomials evaluated at $x = r$ will be 0 modulo p, and thus every polynomial corresponding to a vector in the lattice has this property. As in the previous example, each column is scaled by a power of R, so that the ℓ_1 norm of any vector in this lattice is an upper bound on the value of the corresponding (un-scaled) polynomial evaluated at r.

If we can find a vector in the lattice of length less than p, then it corresponds to a polynomial g that must satisfy $g(r) < p$. Since by construction, $g(r) = 0 \pmod{p}$, this means that $g(r) = 0$ over the integers.

We compute the determinant of the lattice to verify that it contains a sufficiently small vector. For this example, $\det B = R^3 N$. This means we need $(\det B)^{1/\dim L} = (R^3 N)^{1/3} < p$. Solving for R, this gives $R < p^{1/3}$. For an RSA modulus we have $p \approx N^{1/2}$, and thus we get the condition $R < N^{1/6}$.

Question 83. *Up to what limit does this method work?*

This method works up to $R < p^{1/2}$ at the limit by increasing the dimension of the lattice. This is accomplished by taking higher multiples of f and N. Howgrave-Graham's dissertation [HG98] and May's survey [May10] provide details on how to do this.

6.2.2.3 RSA key recovery from least significant bits of p

It is also straightforward to adapt this method to deal with a contiguous chunk of unknown bits in the least significant bits of p. If the chunk begins at bit position ℓ, the input polynomial will have the form $f(x) = 2^\ell x + a$. This can be multiplied by $2^{-\ell} \pmod{N}$ and solved exactly as above.
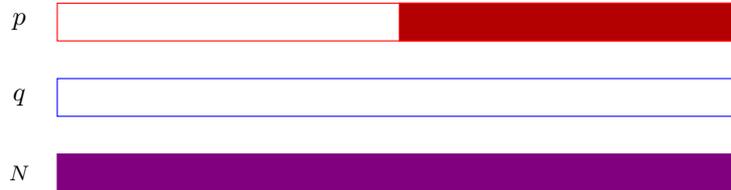
Figure 6.3: Factorization of $N = pq$ given contiguous known least significant bits of p.

6.2.2.4 RSA key recovery from middle bits of p

RSA key recovery from middle bits of p is somewhat more complex than the previous examples, because there are two unknown chunks of bits in the most and least significant bits of p.
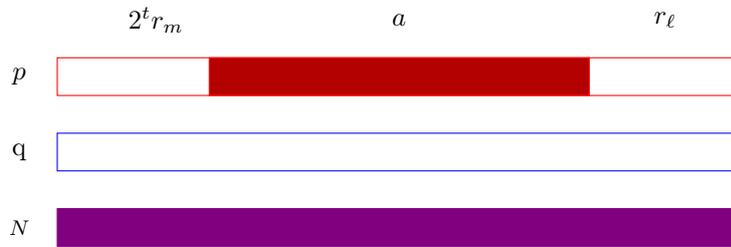
Figure 6.4: Factorization of $N = pq$ given contiguous known bits of p in the middle.

Problem setup. Assume we know a large contiguous portion of the middle bits of p, so that $p = a + r_\ell + 2^t r_m$, where a is an integer representing the known bits of p, and r_ℓ and r_m are unknown integers representing the least and most significant bits of p that we wish to solve for. The value t is the starting bit position of the unknown most significant bits. We know that $|r_\ell| < R$ and $|r_m| < R$ for some bound R.

As a concrete example, let

$$N = \texttt{0x3ab05d0c0694c6bd8ee9683d15039e2f738558225d7d37f4a601bcb9}$$
$$\texttt{29ccfa564804925679e2f3542b}.$$

be a 326-bit RSA modulus. Let

$$a = \texttt{0xc48c998771f7ca68c9788ec4bff9b40b80000}.$$

be the middle bits of one of its factors p. There are 16 unknown bits in the most and least significant bit positions. Thus we know that $R = 2^{16}$ in our concrete example. We wish to recover p.

Cast the problem as finding solutions to a polynomial. In the previous examples, we only had one variable to solve for. Here, we have two, so we need to use a bivariate polynomial. We can write down

$$f(x, y) = x + 2^t y + a,$$

so that $f(r_\ell, r_m) = p$.

In our concrete example, the factor p has 164 bits, so we have $f(x, y) = x + 2^{148}y + a$. We hope to construct two polynomials $g_1(x, y)$ and $g_2(x, y)$ satisfying $g_1(r_\ell, r_m) = 0$ and $g_2(r_\ell, r_m) = 0$ over the integers. Then we can solve the system for the simultaneous roots.

Construct a lattice. As before, we will use our input polynomial f and the public RSA modulus N to construct a lattice. Unfortunately for the simplicity of our example, the smallest polynomial that is guaranteed to result in a nontrivial bound on the solution size for our desired roots has degree 3, and results in a lattice of dimension 10.

As before, each column corresponds to a monomial that appears in our polynomials, and each row corresponds to a polynomial that evaluates to 0 mod p at our desired solution. In our example, we will use the polynomials $f^3, f^2y, fy^2, y^3N, f^2, fy, y^2N, f, yN$, and N. The monomials in the columns are $x^3, x^2y, xy^2, y^3, x^2, xy, y^2, x, y$, and 1. Each column is scaled by the appropriate power of R.

$$
B = \begin{pmatrix}
R^3 & 3 \cdot 2^t R^3 & 3 \cdot 2^{2t} R^3 & 2^{3t} R^3 & 3aR^2 & 6 \cdot 2^t aR^2 & 3 \cdot 2^{2t} aR^2 & 3a^2R & 3 \cdot 2^t a^2 R & a^3 \\
0 & R^3 & 2 \cdot 2^t R^3 & 2^{2t} R^3 & 0 & 2aR^2 & 2 \cdot 2^t aR^2 & 0 & a^2R & 0 \\
0 & 0 & R^3 & 2^t R^3 & 0 & 0 & aR^2 & 0 & 0 & 0 \\
0 & 0 & 0 & R^3N & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & R^2 & 2 \cdot 2^t R^2 & 2^{2t} R^2 & 2aR & 2 \cdot 2^t aR & a^2 \\
0 & 0 & 0 & 0 & 0 & R^2 & 2^t R^2 & 0 & aR & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & R^2N & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & R & 2^t R & a \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & RN & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & N
\end{pmatrix}.
$$

We reduce this matrix using the LLL algorithm, and reconstruct the bivariate polynomials corresponding to each row of the reduced basis. Unfortunately, these are too large to fit on a page.

Solve the system of polynomials to find common roots. Heuristically, we would hope to only need two sufficiently short vectors and then compute the resultant of the corresponding polynomials or use a Gröbner basis to find the common roots, but in our example the two shortest vectors are not algebraically independent. In this case it suffices to use the first three vectors. Concretely, we construct an ideal over the ring of bivariate polynomials with integer coefficients whose basis is the polynomials corresponding to the three shortest vectors in the reduced basis for $\mathcal{L}(B)$ above, and then call a Gröbner basis algorithm on it. For this example, the Gröbner basis is exactly the polynomials $(x - \texttt{0x339b}, y - \texttt{0x5a94})$, which reveals the desired solutions for $x = r_\ell$ and $y = r_m$.

In this example, the nine shortest vectors all vanish at the desired solution, so we could have constructed our Gröbner basis from other subsets of these short vectors.

> **Question 84.** *Why does this work?*

The determinant of our lattice is $\det B = R^{20}N^4$, and the lattice has dimension 10. We hope to find two vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ of length approximately $\det B^{1/\dim B}$. This is not guaranteed to be possible, but for random lattices we expect the lengths of the vectors in a reduced basis to have close to the same lengths. The ℓ_1 norms of the vectors $\mathbf{v}_1$ and $\mathbf{v}_2$ are upper bounds on the magnitude of the corresponding polynomials $f_{\mathbf{v}_1}(x, y)$, $f_{\mathbf{v}_2}(x, y)$ evaluated at the desired roots r_ℓ, r_m. In order to guarantee that these vanish, we want the inequality

$$
|f_{\mathbf{v}_i}(r_\ell, r_m)| \le ||\mathbf{v}_i||_1 < p \approx \sqrt{N}
$$

to hold.

Thus the desired condition for success is

$$
\det B^{1/\dim B} < \sqrt{N},
$$
$$
\left(R^{20}N^4\right)^{1/10} < N^{1/2},
$$
$$
R^{20} < N.
$$

In our example, N was 326 bits long, and we chose R to have 16 bits.

This attack was applied in [BCC+13] to recover RSA keys generated by a faulty random number generator that generated primes with predictable sequences of bits.

6.2.2.5 RSA key recovery from multiple chunks of bits of p

The above idea can be extended to handle more chunks of p at the cost of increasing the dimension of the lattice. Each unknown "chunk" of bits introduces a new variable in the linear equation that will be solved for p. At the limit, the algorithm requires 70% of the bits of p divided into at most $\log \log N$ blocks [HM08].
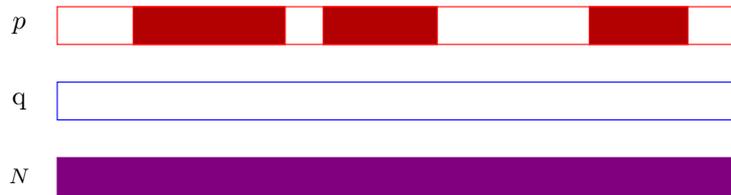
Figure 6.5: Factorization of $N = pq$ given multiple chunks of p.

6.2.2.6 Open problem: RSA key recovery from many nonconsecutive bits of p

The above methods scale poorly with the number of chunks of known bits. It is an open problem to develop a sub-exponential time method to recover an RSA key or factor the RSA modulus N with more than $\log \log N$ unknown chunks of bits, if these bits are only known about, say, one factor p of N. If information is known about both p and q or other fields of the RSA private key, then the methods of Section 6.2.3.1 may be applicable.

Open Question 5. *Can we recover an RSA key or factor an RSA modulus N with more than $\log \log N$ unknown chunks of bits coming from one factor of N?*

Figure 6.6: Efficient factorization of $N = pq$ given many chunks of p and no information about q is an open problem.

6.2.2.7 Partial recovery of RSA d_p

Recovering the CRT coefficient $d_p = d \pmod{p-1}$ from a large contiguous chunk of bits can be done using the approach given in Section 6.2.2.2. We illustrate the method in the case of known most significant bits.
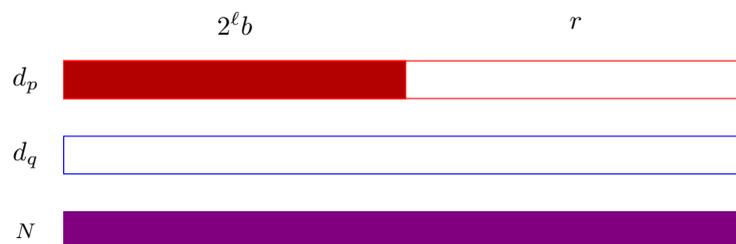
Recovering RSA $d_p = d \pmod{(p-1)}$ given many contiguous bits of d_p.

Problem setup. Let

$$N = \texttt{0x4d14933399708b4a5276373cb5b756f312f023c43d60b323ba24cee670f5}$$

be a 240-bit RSA modulus. We will use public exponent $e = 65537$.

In this problem, we are given some of the most significant bits b of d_p, and we want to recover the rest. As before, let ℓ be the number of least significant bits of d_p we need to recover, so that there is some value $a = 2^\ell b$ with $a + r = d_p$ for some $r < 2^\ell$. For our concrete example, we have

$$a = \text{0x25822d06984a06be5596fcc0000000}.$$

Cast the problem as finding the roots of a polynomial We start with the relation $ed_p \equiv 1 \pmod{p-1}$ and rewrite it as an integer relation by introducing a new variable k_p as

$$ed_p = 1 + k_p(p-1). \tag{6.4}$$

The integer k_p is unknown, but we know that $k_p < e$ since $d_p < (p-1)$. In our example, and typically in practice, we have $e = 65537$, so we will run the attack for all possible values of $1 \le k_p < 65537$. With the correct parameters, we are guaranteed to find a solution for the correct value of k_p. For other incorrect guesses of k_p, in practice the attack is unlikely to result in any solutions found, but any spurious solutions that arise can be eliminated because they will not result in a factorization of N.

We can rearrange Equation 6.4, with e^{-1} computed modulo N such as

$$e(a + r) - 1 + k_p \equiv 0 \pmod{p},$$
$$a + r + e^{-1}(k_p - 1) \equiv 0 \pmod{p}.$$

Let $A = a + e^{-1}(k_p - 1)$. Then we wish to find a small root r of the polynomial $f(x) = A + x$ modulo p, where $|r| < R$. For our concrete example, we have $R = 2^{30}$ and $k_p = 23592$, so

$$A = \text{0x8ffe9143aa4c189787058057a0784576848f3f28d79a83169f72a0550699112}.$$

Construct a lattice. Since the form of the problem is identical to the previous section, we use the same lattice construction

$$B = \begin{pmatrix} R^2 & RA & 0 \\ 0 & R & A \\ 0 & 0 & N \end{pmatrix}.$$

We apply the LLL algorithm to this basis and take the shortest vector in the reduced basis. For our example, this is

$$\mathbf{v} = (-\text{1306dd0a37ec}R^2, \text{52955e433295de64273}R, -\text{31db63ed6f29f4d8f4d1501c47}).$$

We construct the corresponding polynomial

$$f(x) = -\text{1306dd0a37ec}x^2 + \text{52955e433295de64273}x - \text{31db63ed6f29f4d8f4d1501c47}.$$

Computing the roots of f, we discover that $r = \text{0x39d9b141}$ is among them, and that $\gcd(A + r, N) = p$.

> **Question 85.** *Up to what limit does this method work?*

At the limit, this technique can work up to $R < p^{1/2}$ [BM03] by increasing the dimension of the lattice with higher degree polynomials and higher multiplicities of the root.

6.2.2.8 Partial recovery of RSA d from most significant bits is not possible

Partial recovery for d varies somewhat depending on the bits that are known and the size of e. Since e is small in practice, we will focus on that case here.

Figure 6.7: For small exponent e, the most significant bits of d do not allow full key recovery.

Most significant bits of d. When e is small enough to brute force, the most significant half of bits of d can be recovered easily with no additional information. This implies that if full key recovery were possible from only the most significant half of bits of d, then small public exponent RSA would be completely broken. Since small public exponent RSA is not known to be insecure in general, this unfortunately means that no such key recovery method is possible for this case.

Consider the RSA equation

$$
\begin{aligned}
ed &= 1 \quad (\bmod \ (p-1)(q-1)), \\
ed &= 1 + k(p-1)(q-1), \\
ed &= 1 + k(N - (p+q) + 1), \\
d &= kN/e - (k(p+q-1) - 1)/e.
\end{aligned}
$$

Since $p + q \approx \sqrt{N}$, the second term affects only the least significant half of the bits of d, so the value kN/e shares approximately the most significant half of its bits in common with d.

On the positive side, this observation allows the attacker to narrow down possible values for k if the attacker knows any most significant bits of d for certain. See Boneh, Durfee, and Frankel [BDF98] for more details.

6.2.2.9 Partial recovery of RSA d from least significant bits

For low-exponent RSA, if an adversary knows the least significant t bits of d, then this can be transformed into knowledge of the least significant t bits of p, and then the method of Section 6.2.2.3 can be applied. This algorithm is due to Boneh, Durfee, and Frankel [Bon98].

Figure 6.8: Recovering RSA p given contiguous least significant bits of d.

Assume the adversary knows the t least significant bits of d; call this value d_0. Then

$$ ed_0 \equiv 1 + k(N - (p+q) + 1) \quad (\bmod \ 2^t). $$

Let $s = p + q$. The adversary tries all possible values of k, $1 < k < e$ to obtain e candidate values for the t least significant bits of s. Then for each candidate s, the least significant bits of p are solutions to the quadratic equation

$$ p^2 - sp + N \equiv 0 \quad (\bmod \ 2^t). $$

Let a be a candidate solution for the least significant bits of p. Putting this in the context of Section 6.2.2.3, the attacker wishes to solve $f(x) = a + 2^t x \equiv 0 \pmod{p}$. This can be multiplied by $2^{-t} \pmod{N}$ and the exact method of Section 6.2.2.3 can be applied to recover p. Since at the limit, the methods of Section 6.2.2.3 work to recover $N^{1/4}$ bits of p, this method will work when as few as $N^{1/4}$ bits of d are known.

There are more sophisticated lattice algorithms that involve different tradeoffs, but for very small e, which is typically the case in practice, they require nearly all of the least significant bits of d to be known [BM03].

6.2.3 Non-consecutive bits known with redundancy

This section covers key recovery in the case that many non-consecutive bits of secret values are known or need to be recovered. The lattice methods covered in the previous section can be adapted to recover multiple chunks of unknown key bits, but at a high cost. The lattice dimension increases with the number of chunks, and when a large number of bits is to be recovered, the running time can be exponential in the number of chunks.

In this section, we explore a different technique that allows a different tradeoff. In this case, the attacker has knowledge of many non-contiguous bits of secret key values, and knows these for multiple secret values of the key. The attacker might have learned parts of both p and q, or $d \pmod{p-1}$ and $d \pmod{q-1}$, for example.

6.2.3.1 Random known bits of p and q

Figure 6.9: Factorization of $N = pq$ given non-consecutive bits of both p and q.

We begin by analyzing a case that is less likely to arise in practice, the case of random erasures of bits of p and q, in order to give the main ideas behind the algorithm in the simplest setting.

The main technique used for these cases is a branch and prune algorithm. The idea behind the branch and prune algorithm is to write down an integer relationship between the elements in the secret key and the public key, and progressively solve for unknown bits of the secret key, starting at the least significant bits. This produces a tree of solutions. Every branch corresponds to guesses for one or more unknown bits at a particular solution, and branches are pruned if the guesses result in incorrect relationships to the public key. This algorithm is presented and analyzed in [HS09].

Problem setup. Let $N = 899$. Imagine we have learned some bits of p and q, in an erasure model. For each bit position, we either know the bit value, or we know that we do not know it. For example, we have

$$p = \sqcup 11 \sqcup 1,$$

and

$$q = \sqcup 1 \sqcup 0 \sqcup.$$

Defining an integer relation. The integer relation that we will take advantage of for this example is $N = pq$.

Iteratively solve for each bit. The main idea of the algorithm is to iteratively solve for the bits of the unknowns p and q, starting at the least significant bits. These can then be checked against the known public value of N.

At the least significant bit, the value is known for p and is unknown for q. There are two options for the value of q, but only the bit value 1 satisfies the constraint that $pq = N \pmod 2$. The algorithm then proceeds to the next step, where the value of the second bit is known for q but not for p. Only the bit value 1 satisfies the constraint $pq = N \pmod{2^2}$, so the algorithm continues down this branch. Since this generates a tree, the tree can be traversed in depth-first or breadth-first order; depth-first will be more memory efficient. This is illustrated in Figure 6.10.
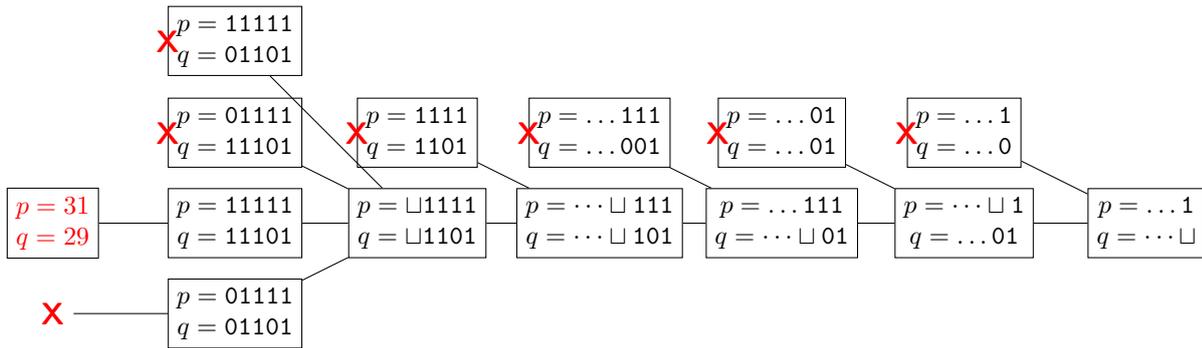
Figure 6.10: The branch and prune tree for our numeric example. The algorithm begins at the right-hand node representing the least significant bits, and iteratively branches and prunes guesses for successive bits moving towards the most significant bits.

The algorithm works because $N = pq \pmod{2^i}$ for all values of i. Additionally, we want some assurance that an incorrect guess for a value at a particular bit location should eventually lead to that branch being pruned. Heuristically, when the ith bits of both p and q are unknown, the tree will branch. When bit i is known for one but not the other, there will be a unique solution. When the ith bits of both p and q are known, an incorrect solution has around a 50% probability of being pruned. Thus the algorithm is expected to be efficient as long as there are not long runs of simultaneous unknown bits. We assume the length of p and q is known. Once the algorithm has traversed this many bits, the final solution $pq = N$ can be checked without modular constraints.

When random bits are known from p and q, the analysis of [HS09] shows that the tree of generated solutions is expected to have polynomial size when 57% of the bits of p and q are revealed at random. This algorithm can still be efficient if the distribution of bits known is not random, as long as it allows efficient pruning of the tree. An example would be learning 3 out of every 5 bits of p and q, as in [YGH16].

Paterson, Polychroniadou, and Sibborn [PPS12] give an analysis of the required information for different scenarios, and observe that doing a depth-first search is more efficient memory-wise than a breadth-first search.

6.2.3.2 Random known bits of the Chinese remainder coefficients $d \pmod{p-1}$ and $d \pmod{q-1}$.

The description in Section 6.2.3.1 can be extended to recover the Chinese remainder exponents $d_p = d \pmod{p-1}$ and $d_q = d \pmod{q-1}$ using the same technique as the previous section. This is the most common case encountered in RSA side-channel attacks.
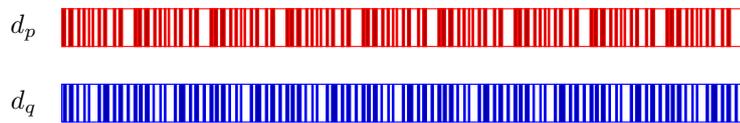
Factorization of $N = pq$ given non-consecutive bits of d_p, d_q.

Problem setup. Let $N = 899$ be the RSA public modulus, and $e = 17$ be the public exponent. Imagine that the adversary has recovered some bits of the secret Chinese remainder exponents $d_p = d \pmod{p-1}$ and $d_q = d \pmod{q-1}$,

$$d_p = \sqcup 0 \sqcup \sqcup 1, \qquad d_q = \sqcup \sqcup \sqcup 0 \sqcup$$

We wish to recover the missing unknown bits of d_p and d_q, which will allow us to recover the secret key itself.

Define integer relations.   We know that $ed_p \equiv 1 \pmod{p-1}$ and $ed_q \equiv 1 \pmod{q-1}$. We rewrite these as integer relations

$$ed_p = 1 + k_p(p-1), \qquad ed_q = 1 + k_q(p-1).$$

We have no information about the values of p and q, but their values are uniquely determined from a guess for d_p or d_q. We also know that

$$pq = N.$$

The values k_p and k_q are unknown, so we must brute force them by running the algorithm for all possible values. We expect it to fail for incorrect guesses, and succeed for the unique correct guess. Equation 6.2 in Section 6.2.1 shows that there is a unique value of k_q for a given guess for k_p. Since $k_p < e$ we need to brute force at most e pairs of values for k_p and k_q. In our example, we have $k_p = 13$ and $k_q = 3$, although this won't be verified as the correct guesses until the solution is found.

Iteratively solve for each bit.   With our integer relations in place, we can then use them to iteratively solve for each bit of the unknowns d_p, d_q, p, and q, starting from the least significant bit. We check guesses for each value against our three integer relations, and at bit i we prune those that do not satisfy the relations mod 2^i. We have three relations and four unknowns, so we generate at most two new branches at each bit.

$$ed_p - 1 + k_p \equiv k_p p \pmod{2^i},$$
$$ed_q - 1 + k_q \equiv k_q q \pmod{2^i},$$
$$pq \equiv N \pmod{2^i}.$$

Since the values of p and q up to bit i are uniquely determined by our guess for d_p and d_q up to bit i, the algorithm prunes solutions based on the relation $pq \equiv N \pmod{2^i}$. The analysis of this case is then identical to the case of learning bits of p and q at random.

For incorrect guesses for the values of k_p and k_q, we expect the equations to act like random constraints, and thus to quickly become unsatisfiable. Once there are no more possible solutions in a tree, the guess for k_p and k_q is known to be incorrect. This is illustrated by Figure 6.11.

6.2.3.3   Recovering RSA keys from indirect information

For this type of key recovery algorithm, it is not always necessary to have direct knowledge of bits of the secret key values with certainty. It can still be possible to apply the branch-and-prune technique to recover secret keys even if only "implicit" information is known about the secret values, as long as this implicit information implies a relationship that can be checked to prioritize or prune candidate key guesses from the least significant bits. Examples in the literature include [BBG$^+$17], which computes partial sliding window square-and-multiply sequences for candidate guesses and compares them to the ground truth measurements, and [MVH$^+$20], which compares the sequence of program branches in a binary GCD algorithm implementation computed over the cryptographic secrets to a ground truth measurement.

6.2.3.4   Open problem: Random known bits without redundancy

Open Question 6. *Can we efficiently recover an RSA secret key when random bits of a factor of N are known without redundancy?*

As mentioned in Section 6.2.2.6, it is an open problem to recover an RSA secret key when many nonconsecutive chunks of bits need to be recovered, and the bits known are from only one secret key field, with no additional information from other values. Applying the branch-and-prune methods discussed in this section to a single secret key value, say a factor p of N, where random bits are known, would result in a tree with exponentially many solutions unless additional information were available to prune the tree.

6.3   Key recovery methods for DSA and ECDSA

Now that we have presented the key recovery methods for RSA, we move on to the DSA and its elliptic curve variant ECDSA. The methods used in this context are quite similar in their core ideas. We start by presenting the protocols.

Figure 6.11: We give a sample branch and prune tree for recovering d_p and d_q from known bits, starting from the least significant bits on the right side of the tree. At each bit location, the value of p up to bit i is uniquely determined by the guess for d_p up to bit i, and the value of q up to bit i is uniquely determined by the guess for d_q up to bit i. The red X marks the branches that are pruned by verifying the relation $pq = N \pmod{2^i}$.

6.3.1 DSA and ECDSA preliminaries

From the perspective of partial key recovery, DSA and ECDSA are very similar, and we will cover them together. We will use slightly nonstandard notation to describe each signature scheme to make them as close as possible, so that we can use the same notation to describe the attacks simultaneously.

6.3.1.1 DSA

The Digital Signature Algorithm [NIS13] (DSA) is an adaptation of the ElGamal Signature Scheme [ElG85] that reduces the amount of computation required and the resulting signature size by using Schnorr groups [Sch90].

Parameter Generation. A DSA public key includes several global parameters specifying the group to work over: a prime p, a subgroup of order n satisfying $n \mid (p-1)$, and an integer g that generates a group of order $n \pmod{p}$, where n is typically much smaller than p, for example 256 bits for a 2048-bit p. A single set of group parameters can be shared across many public keys, or individually generated for a given public key. To generate a long-term private signing key, an implementation starts by choosing the secret key $0 < \alpha < n$ and computing $y = g^\alpha \pmod{p}$. The public key is the tuple (y, g, p, n) and the private key is (α, g, p, n).

Signature Generation. To sign a message m, implementations apply a collision-resistant hash function H to m to obtain a hashed message $h = H(m)$. To generate the signature, the implementation generates an ephemeral secret integer $0 < k < n$, and computes the integers $r = g^k \pmod{p} \pmod{n}$, and $s = k^{-1}(h + \alpha r) \pmod{n}$. The signature is the pair (r, s).

6.3.1.2 ECDSA

The Elliptic Curve Digital Signature Algorithm (ECDSA) is an adaptation of DSA to use elliptic curves instead of Schnorr groups.

Parameter Generation. An ECDSA public key includes global parameters specifying an elliptic curve E over a finite field together with a generator point g of a subgroup over E of order n. To generate a long-term private signing key, an implementation starts by choosing a secret integer $0 < \alpha < n$, and computing the elliptic curve point $y = \alpha g$ on E. The public key is the elliptic curve point y together with the global parameters specifying E, g, and n. The private key is the integer α together with these global parameters.

Signature Generation. To sign a message m, implementations apply a collision-resistant hash function H to m to obtain a hashed message $h = H(m)$. To generate the signature, the implementation generates an ephemeral secret $0 < k < n$. The implementation computes the elliptic curve point kg and sets the value r to be the x-coordinate of kg. The implementation then computes the integer $s = k^{-1}(h + \alpha r) \pmod{n}$. The signature is the pair of integers (r, s).

6.3.1.3 Nonce recovery and (EC)DSA security.

The security of (EC)DSA is extremely dependent on the signature nonce k being securely generated, uniformly distributed, and unique for every signature. If the nonce for one or more signatures is generated in a vulnerable manner, then an attacker may be able to efficiently recover the long-term secret signing key. Because of this property, side-channel attacks against (EC)DSA almost universally target nonce generation.

Key recovery from signature nonce. For a DSA or ECDSA key, if the nonce k is known for a single signature, it is simple to compute the long-term private key. Rearranging the expression for s, the secret key α can be recovered as

$$\alpha = r^{-1}(ks - h) \pmod{n}. \tag{6.5}$$

6.3.2 (EC)DSA key recovery from most significant bits of the nonce k

There are two families of techniques for (EC)DSA key recovery from most significant bits of the nonce k. Both techniques require knowing information about the nonce used in multiple signatures from the same secret key. We assume that the attacker knows the long-term public signature verification key, and

has access to multiple signatures generated using the corresponding secret signing key. The attacker also needs to know the hash of the messages that the signatures correspond to.

Figure 6.12: (EC)DSA key recovery from signatures where most significant bits of the nonces are known.

The first technique is via lattices. This is generally considered more straightforward to implement, and works well when more nonce bits are known, and information from fewer signatures is available. We would need to know at least two most significant bits from the nonces of dozens to hundreds of signatures. We cover this technique below.

The second technique is via Fourier analysis [Ble98, DHMP13] originally due to Bleichenbacher. This technique can deal with as little as one known most significant bit from signature nonces, but empirically appears to require an order of magnitude or more signatures than the lattice approach, and as many as 2^{32}–2^{35} for record computations [ANT+20]. We leave a more detailed discussion of this technique for future work. Nice descriptions of the algorithm can be found in [DHMP13, TTA18].

6.3.2.1 Lattice attacks

The main idea behind lattice attacks for (EC)DSA key recovery is to formulate the (EC)DSA key recovery problem as an instance of the *Hidden Number Problem* and then compute the shortest vector of a specially constructed lattice to reveal the solution.

The Hidden Number Problem The Hidden Number Problem was introduced by Boneh and Venkatesan [BV96] to show that the most significant bits of a Diffie-Hellman shared secret are hardcore, meaning they are as hard to compute as computing the secret key itself. Nguyen and Shparlinski showed how to use this approach to break DSA and ECDSA from information about the nonces [NS02, NS03]. Various extensions of the technique can deal with different numbers of bits known per signature [BvSY14] or errors [DDME+18].

Below we give a simplified example that shows how to recover the key from a small number of signatures when many of the most significant bits of the nonce are zero, and then we will show how to extend the attack to more signatures with fewer bits known from each nonce, and cover the case of arbitrary bits known from the nonce.

Problem setup. Let $p = \mathtt{0xfffffffffffffd21f}$ be a 64-bit prime, and let $E : y^2 = x^3 + 3$ be an elliptic curve over $\mathbb{F}_p$. Let $g = (1, 2)$ be our generator point on E, which has order $n = \mathtt{0xfffffffefa23f437}$. We have two ECDSA signatures

$$(r_1, s_1) = (\mathtt{6393e79fbfb40c9c}, \mathtt{621ee64e65d1e938})$$
$$\text{on message hash} \quad h_1 = \mathtt{ae0f1d8cd0fd6dd1},$$

and

$$(r_2, s_2) = (\mathtt{3ea8720afa6d03c2}, \mathtt{16fc6aa65bf241ea})$$
$$\text{on message hash} \quad h_2 = \mathtt{8927e246fe4f3941}.$$

These signatures both use 32-bit nonces k; that is, we know that their 32 most significant bits are 0.

156

Cast the problem as a system of equations. Our signatures above satisfy the equivalencies

$$s_1 \equiv k_1^{-1}(h_1 + \alpha r_1) \pmod{n},$$
$$s_2 \equiv k_2^{-1}(h_2 + \alpha r_2) \pmod{n}.$$

The values k_1, k_2, and α are unknown; the other values are known. We can eliminate the variable α and rearrange terms as follows:

$$k_1 - s_1^{-1}s_2 r_1 r_2^{-1}k_2 + s_1^{-1}r_1 h_2 r_2^{-1} - s_1^{-1}h_1 \equiv 0 \pmod{n}.$$

Let $t = -s_1^{-1}s_2 r_1 r_2^{-1}$ and $u = s_1^{-1}r_1 h_2 r_2^{-1} - s_1^{-1}h_1$. We can then simplify the above as

$$k_1 + tk_2 + u \equiv 0 \pmod{n}. \tag{6.6}$$

We wish to solve for k_1 and k_2, and we know that they are both small. Let $|k_1|, |k_2| < K$. For our example, we have $K = 2^{32}$.

Construct a lattice. We construct the following lattice basis

$$B = \begin{pmatrix} n & 0 & 0 \\ t & 1 & 0 \\ u & 0 & K \end{pmatrix}.$$

The vector $\mathbf{v} = (k_1, k_2, K)$ is in this lattice by construction, and we expect it to be particularly short. Calling the LLL algorithm (or BKZ algorithm) on the basis B results in a basis that contains this short vector

$$\mathbf{v} = (-\texttt{0x270feca3}, \texttt{0x4dbd2db0}, \texttt{0x100000000})$$

as the third vector in the reduced basis. We can verify that the value r_1 in our example matches the x-coordinate of $k_1 g$, and we can use Equation 6.5 to compute the private key α.

> **Question 86.** *Why does this work?*

In our example, we have constructed a lattice that is guaranteed to contain our target vector. In order for this method to work, we hope that it is the shortest vector, or close to the shortest vector in the lattice, and we solve the unique shortest vector problem in the lattice in order to find it.

The vector $\mathbf{v} = (k_1, k_2, K)$ has length $||\mathbf{v}||_2 \leq \sqrt{3}K$ by construction. Our lattice has determinant $\det B = nK$. Ignoring constants for the moment, if our lattice were truly random, we would expect the shortest vector to have length $\approx \det B^{1/\dim B}$. Thus if $||\mathbf{v}||_2 < \det B^{1/\dim B}$, we expect it to be the shortest vector in the lattice, and to be found by a sufficiently good approximation to the shortest vector problem. For our example, we expect this to be satisfied when $K < (nK)^{1/3}$, or when $K < \sqrt{n}$.

The way we have presented this method may remind the reader of the flavor of the methods in Section 6.2.2.1. The specific lattice construction used here is a sort of "dual" to the constructions from Section 6.2.2.1, in that the target vector is the desired solution to our system of equations. However, in contrast to Section 6.2.2.1, we are not guaranteed to find the solution we desire once we find a sufficiently short vector: this method can fail with probability that decreases the shorter our target vector α is compared to the expected shortest vector length.

Scaling to many signatures to decrease the number of bits known. To decrease the number of bits required from each signature, we can incorporate more signatures into the lattice. If we have access to many signatures $(r_1, s_1), \ldots, (r_m, s_m)$ on message hashes $h_1, \ldots, h_m$, we use the same method above to write down equivalencies $s_i \equiv k_i^{-1}(h_i + \alpha r_i) \pmod{n}$, then as above we rearrange terms and eliminate the variable α to obtain

$$k_1 + t_1 k_m + u_1 \equiv 0 \pmod{n}$$
$$k_2 + t_2 k_m + u_2 \equiv 0 \pmod{n}$$
$$\vdots \tag{6.7}$$
$$k_{m-1} + t_{m-1} k_m + u_{m-1} \equiv 0 \pmod{n}$$

We then construct the lattice

$$
B = \begin{pmatrix}
n & & & & & \\
& n & & & & \\
& & \ddots & & & \\
& & & n & & \\
t_1 & t_2 & \ldots & t_m & 1 & \\
u_1 & u_2 & \ldots & u_m & 0 & K
\end{pmatrix}.
$$

In order to solve SVP, we must run an algorithm like BKZ with block size $\dim \mathcal{L}(B) = m + 2$. Using BKZ to look for the shortest vector can be done relatively efficiently up to dimension around 100 currently; beyond that it becomes increasingly expensive. In practice, one can often achieve a faster running time for fixed parameters by using more samples to construct a larger dimension lattice, and applying BKZ with a smaller block size to find the target vector. This method can recover a secret key from knowledge of the 4 most significant bits of nonces from 256-bit ECDSA signatures using about 70 samples, and 3 most significant bits using around 95 samples. For fewer bits known, either the Fourier analysis technique or a more powerful application of these lattice techniques is required, along with significantly more computational power.

Known nonzero most significant bits. If the most significant bits of the k_i are nonzero and known, we can write $k_i = a_i + b_i$, where the a_i are known, and the b_i are small, so satisfy some bound $|b_i| < K$. Then substituting into Equation 6.6, we obtain

$$
\begin{aligned}
(a_i + b_i) + t_i(a_m + b_m) + u_i &\equiv 0 \pmod{n}, \\
b_i + t_i b_m + u_i + a_i + t_i a_m &\equiv 0 \pmod{n}.
\end{aligned}
$$

Thus we can let $u_i' = u_i + a_i + t_i b_m$, and use the same lattice construction as above, with u_i' substituted for u_i.

Nonce rebalancing. The signature nonces k_i take values in the range $0 < k_i < n$, but the lattice construction bounds the absolute value $|k_i|$. Thus if we know that $0 < k_i < K$ for some bound K, we can achieve a tighter bound by renormalizing the signatures. Let $k_i' = k_i - K/2$, so that $|k_i'| < K/2$. Then we can write Equations 6.7 as

$$
\begin{aligned}
k_i + t_i k_m + u_i &\equiv 0 \pmod{n}, \\
(k_i' + K/2 + t_i(k_m' + K/2)) + u_i &\equiv 0 \pmod{n}, \\
k_i' + t_i k_m' + (t_i + 1)K/2 + u_i &\equiv 0 \pmod{n}.
\end{aligned}
$$

Thus we have an equivalent problem with $t_i' = t_i$, $u_i' = (t_i + 1)K/2 + u_i$, and $K' = K/2$, and can solve as before. This optimization can make a significant difference in practice by reducing the number of required samples as we will see for example in Chapter 7.

6.3.2.2 (EC)DSA key recovery from least significant bits of the nonce k

The attack described above works just as well for known least significant bits of the (EC)DSA nonce.

Figure 6.13: (EC)DSA key recovery from signatures where least significant bits of the nonces are known.

Problem setup. We input a collection (EC)DSA signatures (r_i, s_i) on message hashes h_i. For each signature, we know the least significant bits, so the signature nonces k_i satisfy

$$k_i = a_i + 2^\ell b_i,$$

for known a_i, and b_i unknown but satisfying $|b_i| < B$. Substituting these into Equations 6.7, we get

$$a_i + 2^\ell b_i + t_i(a_m + 2^\ell b_m) + u_i \equiv 0 \pmod{n},$$
$$2^\ell b_i + 2^\ell t_i b_m + a_i + t_i a_m + u_i \equiv 0 \pmod{n},$$
$$b_i + t_i b_m + 2^{-\ell}(a_i + t_i a_m + u_i) \equiv 0 \pmod{n}.$$

We have an equivalent instance of the problem with $t_i' = t_i$, $u_i' = 2^{-\ell}(a_i + t_i a_m + u_i)$, and $B' = B$, and solve as above.

6.3.2.3 (EC)DSA key recovery from middle bits of the nonce k
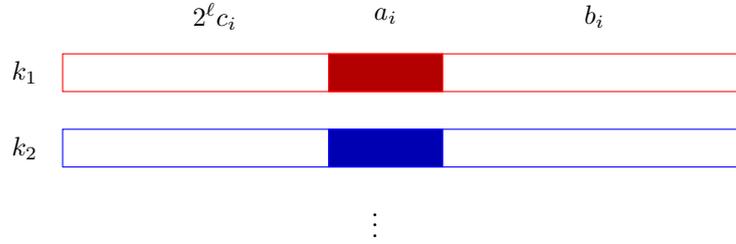
Figure 6.14: (EC)DSA key recovery from signatures where middle bits of the nonces are known.

Recovering an ECDSA key from middle bits of the nonce k is slightly more complex than the methods discussed above, because we have two unknown "chunks" of the nonce to recover per signature. Fortunately, we can deal with these by extending the methods to multiple variables per signature. The method we will use here is similar to the multivariate extension in Section 6.2.2.4, but this case is simpler.

Problem setup. We will use the same elliptic curve group parameters as above. Consider the 64-bit prime $p = \texttt{0xfffffffffffffd21f}$, and let $E : y^2 = x^3 + 3$ be an elliptic curve over $\mathbb{F}_p$. Let $g = (1, 2)$ be our generator point on E, which has order $n = \texttt{0xfffffffefa23f437}$. We have two ECDSA signatures

$$(r_1, s_1) = (\texttt{1a4adeb76b4a90e0}, \texttt{eba129bb2f97f7cd})$$
$$\text{on message hash} \quad h_1 = \texttt{608932fcfaa7785d},$$

and

$$(r_2, s_2) = (\texttt{c4e5bec792193b51}, \texttt{0202d6eecb712ae3})$$
$$\text{on message hash} \quad h_2 = \texttt{4de972930ab4a534}.$$

We know some middle bits of the corresponding nonces. Let

$$a_1 = \texttt{0x50e2fd5d8000}$$

be the middle 34 bits of the signature nonce k_1 used for the first signature above. The first and last 15 bits are unknown. Let

$$a_2 = \texttt{0x172930ab48000}$$

be the middle 34 bits of the signature nonce k_2 used for the second signature above.

159

Cast the problem as a system of equations. As above, our two signature nonces k_1 and k_2 satisfy the relation

$$k_1 + tk_2 + u \equiv 0 \pmod{n}, \tag{6.8}$$

where $t = -s_1^{-1}s_2r_1r_2^{-1}$ and $u = s_1^{-1}r_1h_2r_2^{-1} - s_1^{-1}h_1$. Since we know the middle bits of k_1 and k_2 are a_1 and a_2 respectively, we can write

$$k_1 = a_1 + b_1 + 2^\ell c_1 \quad \text{and} \quad k_2 = a_2 + b_2 + 2^\ell c_2,$$

where b_1, c_1, b_2, and c_2 are unknown but small, less than some bound K. In our example, we have $|b_1|$, $|b_2|$, $|c_1|$, $|c_2| \leq 2^{15}$ and $\ell = 64 - 15 = 49$. Substituting and rearranging into Equation 6.8, we have

$$b_1 + 2^\ell c_1 + tb_2 + 2^\ell tc_2 + a_1 + ta_2 + u \equiv 0 \pmod{n}.$$

Let $u' = a_1 + ta_2 + u$. We wish to find the small solution $x_1 = b_1$, $y_1 = c_1$, $x_2 = b_2$, $y_2 = c_2$ to the linear equation

$$f(x_1, y_2, x_2, y_2) = x_1 + 2^\ell y_1 + tx_2 + 2^\ell ty_2 + u' \equiv 0 \pmod{n}. \tag{6.9}$$

Construct a lattice. We construct the following lattice basis

$$B = \begin{pmatrix} K & K \cdot 2^{49} & Kt & Kt \cdot 2^{49} & u' \\ & Kn & & & \\ & & Kn & & \\ & & & Kn & \\ & & & & n \end{pmatrix}.$$

If we call the LLL or BKZ on B, we obtain a basis that contains the vector

$$\mathbf{v} = (\texttt{0x6589e5fb1823}K, -\texttt{0x42b0986d3e11}K, \texttt{0x8d3b91566f89}K,$$
$$\texttt{0x41be198fb49e}K, -\texttt{0x1dd626d2645d8f7e}).$$

This corresponds to the linear equation

$$\texttt{0x6589e5fb1823}x_1 - \texttt{0x42b0986d3e11}y_1 + \texttt{0x8d3b91566f89}x_2$$
$$+ \texttt{0x41be198fb49e}y_2 - \texttt{0x1dd626d2645d8f7e} = 0.$$

We can do the same for the next three short vectors in the basis, and obtain four linear polynomials in our four unknowns. Solving the system, we obtain the solutions

$$x_1 = \texttt{0x241c}, \quad y_1 = \texttt{0x39a2}, \quad x_2 = \texttt{0x2534}, \quad y_2 = \texttt{0x26f4}.$$

> **Question 87.** *Why does this work?*

The row vectors of the lattice correspond to the weighted coefficient vectors of the linear polynomial f in Equation 6.9, nx_1, ny_1, nx_2, and ny_2. Each of these linear polynomials vanishes by construction modulo n when evaluated at the desired solution $x_1 = b_1$, $y_1 = c_1$, $x_2 = b_2$, $y_2 = c_2$, and thus so does any linear polynomial corresponding to a vector in this lattice. If we can find a lattice vector whose ℓ_1 norm is less than n, then the corresponding linear equation vanishes over the integers when evaluated at the desired solution. In this case, the factorization of n is known, hence we could wonder why it is required to solve over the integers. But HNP is an underdetermined system over n where we have more variables than equations and thus solving mod n would potentially lead to a possibly much larger solution than the one expected. Since we have four unknowns, if we can find four sufficiently short lattice vectors corresponding to four linearly independent equations, we can solve for our desired unknowns.

The determinant of our example lattice is $\det \mathcal{L}(B) = K^4 n^4$, and the lattice has dimension 5. Thus, ignoring approximation factors and constants, we expect to find a vector of length $\det B^{1/\dim B} = (Kn)^{(4/5)}$. This is less than n when $K^4 < n$; in our example this is satisfied because we have chosen a 15-bit K and a 64-bit n.

The determinant bounds guarantee that we will find one short lattice vector, but do not guarantee that we will find four short lattice vectors. For that, we rely on the heuristic assumption that the reduced vectors of a random lattice are close to the same length.

6.3.2.4 (EC)DSA key recovery from many chunks of nonce bits

The above technique can be extended to an arbitrary number of variables.

$$\vdots$$
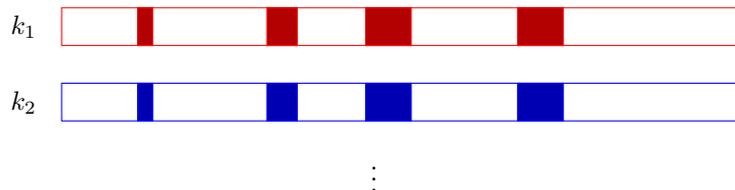
(EC)DSA key recovery from signatures where multiple chunks of the nonces are known.

The extension is called the Extended Hidden Number problem [HR07] and can be used to solve for ECDSA keys when many chunks of signature nonces are known. Each unknown "chunk" of nonce in each signature introduces a new variable, so the resulting lattice will have dimension one larger than the total number of unknowns; if there are m signatures and h unknown chunks of nonce per signature, the lattice will have dimension $mh + 1$. We expect this technique to find the solution when the parameters are such that the system of equations has a unique solution. If the size of each chunk is K, heuristically this will happen when $K^{mh} < n^{m-1}$. This technique has been used in practice in [FWC16] and further explored in [DPP20]. We will discuss the Extended Hidden Number Problem in more details in Chapter 8.

6.4 Key recovery method for the Diffie-Hellman Key Exchange

We now move on to the last protocol we consider in this chapter, the Diffie-Hellman key exchange protocol.

6.4.1 Finite field and elliptic curve Diffie-Hellman preliminaries

The Diffie-Hellman (DH) key exchange protocol [DH76] allows two parties to create a common secret in a secure manner. We summarize the protocol in the context of finite fields and elliptic curves.

Finite field Diffie-Hellman. Finite-field Diffie-Hellman parameters are specified by a prime p and a group generator g. Common implementation choices are p a safe prime, i.e., $q = (p-1)/2$ is prime, in which case g is often equal to $2, 3$ or 4, or p is chosen such that $p - 1$ has a 160, 224, or 256-bit prime factor q and g generates a subgroup of $\mathbb{F}_p^*$ of order q. Key exchange is performed as follows:

1. Alice chooses a random private key a, where $1 \leq a < q$ and computes a public key $A = g^a \pmod{p}$.

2. Bob chooses a random private key b, where $1 \leq b < q$ and computes a public key $B = g^b \pmod{p}$.

3. Alice and Bob exchange the public keys.

4. Alice computes $s_A = B^a \pmod{p}$.

5. Bob computes $s_B = A^b \pmod{p}$.

Because $B^a \pmod{p} = (g^b)^a \pmod{p} = (g^a)^b \pmod{p} = A^b \pmod{p}$, we have $s_A = s_B$. The latter is the secret that now Alice and Bob share.

Elliptic Curve Diffie-Hellman The Elliptic Curve Diffie-Hellman (ECDH) protocol is the elliptic curve counterpart of the Diffie-Hellman key exchange protocol. In ECDH, Alice and Bob agree on an elliptic curve E over a finite field and a generator G of order q. The protocol proceeds as follows:

1. Alice chooses a random private integer a, where $1 \leq a < q$ and computes a public key $A = aG$.

2. Bob chooses a random private integer b, where $1 \leq b < q$ and computes a public key $B = bG$.

3. Alice and Bob exchange the public keys.

4. Alice computes $s_A = aB$.

5. Bob computes $s_B = bA$.

The shared secret is $s_A = aB = a(bG) = b(aG) = bA = s_B$.

6.4.2 Most significant bits of finite field Diffie-Hellman shared secret

The Hidden Number Problem approach we used in the previous section to recover ECDSA or DSA keys from information about the nonces can also be used to recover a Diffie-Hellman shared secret from most significant bits.
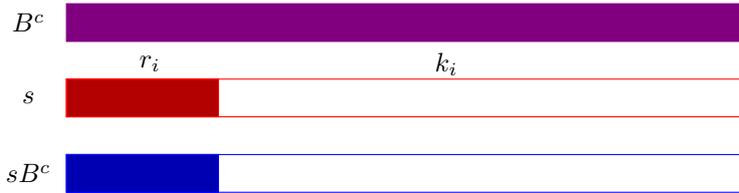
Recovering Diffie-Hellman shared secret from most significant bits of s.

Problem setup. Let $p = \texttt{0xfffffffffffffffffffffffffffffffffc3a7}$ be a 128-bit prime used for finite field Diffie-Hellman, and let $g = 2$ be a generator of the multiplicative group modulo p. Let s the Diffie-Hellman shared secret between public keys

$$A = g^a \pmod p = \texttt{0x3526bb85185259cd42b61e5532fe60e0}$$

and

$$B = g^b \pmod p = \texttt{0x564df0b92ea00ea314eb5a246b01ac9c}.$$

We have learned the value of the first 65 bits of s. Let

$$r_1 = \texttt{0x3330422f6047011b8000000000000000},$$

so we know that $s = r_1 + k_1$ where $k_1 < K = 2^{63}$. Let $c = \texttt{0x56e112dac14f4a4cc02951414aa43a38}$. We have also learned the most significant 65 bits of the Diffie-Hellman shared secret between $AC = g^{a+c} = g^a g^c \pmod p$ and B. Let

$$r_2 = \texttt{0x80097373878e37d20000000000000000}.$$

We know that $g^{(a+c)b} = g^{ab} g^{bc} = sB^c \pmod p$. Let $t = B^c$ so $st = r_2 + k_2 \pmod p$ where $k_2 < K = 2^{63}$.

Cast the problem as a system of equations. We have two relations

$$s = r_1 + k_1 \pmod p, \qquad st = r_2 + k_2 \pmod p,$$

where s, k_1, and k_2 are small and unknown, and r_1, r_2, and t are known. We can eliminate the variable s to obtain the linear equation

$$k_1 - t^{-1}k_2 + r_1 - t^{-1}r_2 \equiv 0 \pmod p.$$

We now have a linear equation in the same form as the Hidden Number Problem we solved in the previous section.

Construct a lattice. We construct the lattice basis

$$B = \begin{pmatrix} p & & \\ t^{-1} & 1 & \\ a_1 - t^{-1}a_2 & & K \end{pmatrix}.$$

If we call the LLL algorithm on the basis B, we obtain a basis that contains the vector

$$(-\texttt{0x2ddb23aa673107bd}, -\texttt{0x216afa75f66a39d5}, \texttt{0x10000000000000000}).$$

This corresponds to our desired solution (k_1, k_2, K), although if the Diffie-Hellman assumption is true, meaning g^{ab} (mod p) looks like a random element of $\mathbb{F}_p^*$, we cannot verify its correctness.

This method is due to Boneh and Venkatesan [BV96], and was the original motivation for their formulation of the Hidden Number Problem. The Raccoon attack recently demonstrated an attack scenario using this technique in the context of TLS [MBA$^+$20].

This method can be adapted to multiple samples with the same number of bits required as the attacks on ECDSA. Knowing the most significant bits of s is not necessary either; we only need the most significant bits of known multiples t_i of s.

6.4.3 Discrete log from contiguous bits of Diffie-Hellman secret exponents

This section addresses the problem of Diffie-Hellman key recovery when the known partial information is part of one or the other of the secret exponents. The technique we apply in this section is Pollard's kangaroo (also known as lambda) algorithm [Pol78] introduced in Chapter 1. Unlike the techniques of the previous sections, which are generally efficient when the attacker's knowledge of the key is above a certain threshold, and either inefficient or infeasible when the attacker's knowledge of the key is below this threshold, this algorithm runs in exponential time, square root of the size of the interval. Thus it provides a significant benefit over brute force, but in practice is likely limited to 80 bits or fewer of key recovery unless you have access to an unusually large amount of computational resources.

Recall that the Pollard kangaroo algorithm is a generic discrete logarithm algorithm that is designed to compute discrete logarithms when the discrete logarithm lies in a small known interval. It applies to both elliptic curve and finite field discrete logarithms. We will use finite field discrete logarithms for our examples, but the algorithm is the same in the elliptic curve context.

6.4.3.1 Known most significant bits of the Diffie-Hellman secret exponent

Problem Setup. Using the same notation for finite fields as in Section 6.4.1, let A be a Diffie-Hellman public key, p be a prime modulus, and g a generator of a multiplicative group of order q modulo p. These values are all public, and thus we assume that they are known. Imagine that we have obtained a consecutive fraction of the most significant bits of the secret exponent a, and we wish to recover the unknown bits of a to reconstruct the secret.
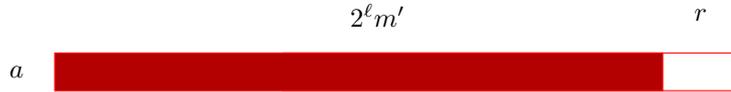
Figure 6.15: Recovering Diffie-Hellman shared secret with most significant bits of secret exponent.

In other words, let $a = m + r$, where $m = 2^\ell m'$ for some known integers m' and ℓ, and $0 \le r < 2^\ell$ is unknown. Let w be the width of the interval that r is contained in: here we have $w = 2^\ell$.

For our concrete example, let $p = \mathtt{0xfef3}$ be a 16-bit prime, and let $g = 3$ be a multiplicative generator of the group of order $q = (p-1)/2 = \mathtt{0x7f79}$ modulo p. We know a Diffie-Hellman public key $A = \mathtt{0xa163}$ and we are given the most significant bits of the secret exponent a but the 8 least significant bits of a are unknown, corresponding to $m = \mathtt{0x1400}$, $\ell = 8$, and $r < 2^8$.

Take some pseudorandom walks. We define a deterministic pseudorandom walk along the following values $s_0, s_1, \ldots, s_i, \ldots$ in our multiplicative group modulo p (and the corresponding exponents $s_0 = g^{x_o}$ (mod p), ..., when known) by choosing a set of random step lengths for the exponents in $[0, \sqrt{w}]$. For our example, we pseudorandomly generated the lengths $(1, 3, 7, 10)$.

$$
s_{i+1} \to \begin{cases} s_i g \pmod{p} & \text{if } s_i \equiv 0 \pmod 4 \\ s_i g^3 \pmod{p} & \text{if } s_i \equiv 1 \pmod 4 \\ s_i g^7 \pmod{p} & \text{if } s_i \equiv 2 \pmod 4 \\ s_i g^{10} \pmod{p} & \text{if } s_i \equiv 3 \pmod 4 \end{cases}
$$

This is a small sample pseudorandom walk generated to run our small example computation. Each step in the pseudorandom walk is determined by the representation of the previous value as an integer $0 \le s_i < p$.

We run two random walks. The first random walk, which is called "the tame kangaroo", starts in the middle of the interval of exponents to be searched, at $s_0 = g^{m + \lfloor \frac{w}{2} \rfloor} \pmod{p}$. In our example, we have $m = \texttt{0x1400}$ and $w = 2^8 = 256$, so the tame kangaroo begins at $s_0 = g^{\texttt{0x1480}} \pmod{p} = \texttt{0x9581}$. We take $\sqrt{w}$ steps along this deterministic pseudorandom path, and store the values s_i together with the exponent x_i that is computed at each step so that $g^{x_i} \equiv s_i \pmod{p}$.

The second random walk is called the "wild kangaroo". It begins at the target $s_0' = A = \texttt{0xa163}$ and follows the same rules as above. We do not know the secret exponent a, but at every step of the walk, we know that $s_i' = Ag^{x_i'} \pmod{p} = g^{a + x_i'} \pmod{p}$. We take at most $\sqrt{w}$ steps along this deterministic pseudorandom path.

If at some point the wild kangaroo's path intersects the tame kangaroo's path, then we are done and can compute the result.
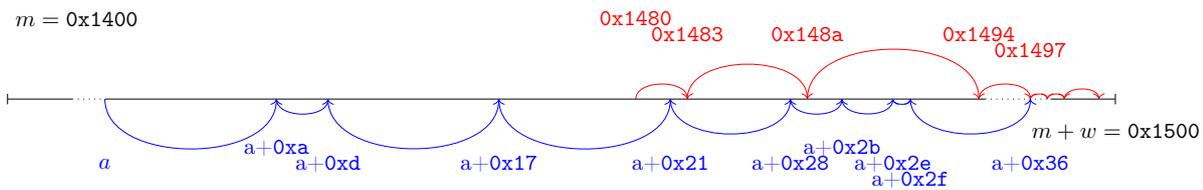
Compute the discrete log. We know that $s_i = s_j'$ for s_i on the tame kangaroo's path and s_j' on the wild kangaroo's path. Thus we have

$$
\begin{aligned}
s_i &= s_j' \pmod{p}, \\
g^{x_i} &= g^{a + x_j'} \pmod{p}, \\
x_i &= a + x_j' \pmod{q}, \\
x_i - x_j' &= a \pmod{q}.
\end{aligned}
$$

In our example, the kangaroos' paths intersected at $g^{\texttt{0x1497}}$ and $g^{a + \texttt{0x36}}$. We can thus compute $a = \texttt{0x1461}$ and verify that $g^{\texttt{0x1461}} \equiv \texttt{0xa163} \pmod{p}$.

Pollard gave the original version of this algorithm in [Pol78]. Teske gives an alternative random walk in [Tes00] that should provide an advantage in theory, but in practice, it seems that no noticeable advantage is gained from it.

We expect this algorithm to reach a collision in $O(\sqrt{w})$ steps; this algorithm thus takes $O(\sqrt{w})$ time to compute a discrete log in an interval of width w. Thus in principle, the armchair cryptanalyst should be able to compute discrete logarithms within intervals of 64 to 80 bits, and those with more resources should be able to go slightly higher than this.

In order to scale to these larger bit sizes, several changes are necessary. First, one typically uses a random walk with many more subdivisions: 32 might be a typical value. Second, van Oorschot and Wiener [OW99] show how to parallelize the kangaroo algorithm using the method of distinguished points as already mentioned in Chapter 2. The idea behind this method is that storing the entire tame kangaroo walk will require too much memory. Instead, one stores a subset of values that satisfy some distinguishing property, such as starting with a certain number of zeros. Then the algorithm launches many wild and tame kangaroo walks, storing distinguished points in a central database. The algorithm is finished when a wild and a tame kangaroo land on the same distinguished point.

Elliptic curves. This algorithm applies equally well to elliptic curve discrete logarithm. One can gain a $\sqrt{2}$ improvement in the complexity of the algorithm as a by-product of the efficiency of inversion on elliptic curves. Since the points P and $-P$ share the same x-coordinate, one can then do a pseudorandom walk on equivalence classes for the relation $P \sim \pm P$.

6.4.3.2 Unknown most significant bits of the Diffie-Hellman secret exponent

Figure 6.16: Recovering Diffie-Hellman shared secret with least significant bits

It is straightforward to extend the kangaroo method to solve for unknown most significant bits of the exponent. As before, we have a known $A = g^a \pmod{p}$ for unknown a that we wish to solve for. In the case of unknown most significant bits, we know an m such that $a = m + 2^\ell r$ for some unknown r satisfying $0 \leq r < w$. The offset ℓ is known. Then we can reduce to the previous problem by running the kangaroo algorithm on the value $A' = g^{2^{-\ell}} A = g^{2^{-\ell} + m + 2^\ell r} \pmod{p}$.

> **Open Question 7.** *Is it possible to recover the Diffie-Hellman secret key with multiple chunks of unknown bits?*
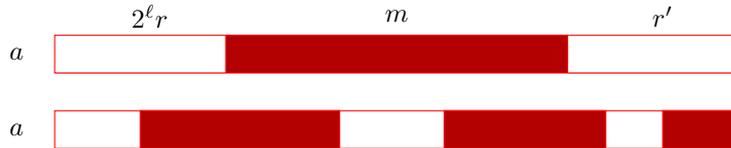
Figure 6.17: Recovering Diffie-Hellman shared secret with multiple chunks of unknown bits.

The case of recovering a Diffie-Hellman secret key in practice with multiple chunks of unknown bits is still an open problem. In theory, finding the secret key in this particular case can be done using a multi-dimensional variant of the discrete logarithm problem. The latter generalizes the discrete logarithm problem in an interval to the case of multiple intervals, see [Rup10, Chapter 6] for further details. In [Rup10], Ruprai analyzes the multi-dimensional discrete logarithm problem for small dimensions. This approach appears to run into boundary issues for multi-dimensional pseudorandom walks when the dimension is greater than five, suggesting that this approach may not extend to the case of recovering many unknown chunks of a Diffie-Hellman exponent.

6.5 Conclusion

This chapter surveyed key recovery methods with partial information for popular public key cryptographic algorithms. We focused in particular on the most widely-deployed asymmetric primitives: RSA, (EC)DSA and Diffie-Hellman. The motivation for these algorithms arises from a variety of side-channel attacks.

While the existence of key recovery algorithms for certain cases may determine whether a particular vulnerability is exploitable or not, we emphasize that these thresholds for an efficiently exploitable key recovery attack should not be used to guide countermeasures. Instead, implementations should strive to have fully constant-time operations for all cryptographic operations to protect against side-channel attacks.

Chapter 7

Cachequote: attacking EPID signature protocol in SGX with HNP

Intel Software Guard Extensions (SGX) allows users to perform secure computation on platforms that run untrusted software. To validate that the computation is correctly initialized and that it executes on trusted hardware, SGX supports attestation providers that can vouch for the user's computation. Communication with these attestation providers is based on the Extended Privacy ID (EPID) protocol, which not only validates the computation but is also designed to maintain the user's privacy. In particular, EPID is designed to ensure that the attestation provider is unable to identify the host on which the computation executes.

In this chapter we investigate the security of the Intel implementation of the EPID protocol. We identify an implementation weakness that leaks information via a cache side channel. We show that a malicious attestation provider can use the leaked information to break the unlinkability guarantees of EPID. We analyze the leaked information using a lattice-based approach for solving the Hidden Number Problem, which we adapt to the zero-knowledge proof in the EPID scheme, extending prior attacks on signature schemes.

This chapter is joint work with Fergus Dall, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom and was published in the proceedings of the CHES 2018 conference [DDME+18]. We notified Intel of our results in January 2018, and the vulnerability has been assigned CVE-2018-3691. Our contribution concerns the key recovery part in Section 7.5.

Contents

7.1   Introduction

Mainstream processors have recently employed Trusted Execution Environments that allow running sensitive computation on potentially-compromised computers owned by untrusted third parties. The most prominent example is Intel Software Guard Extensions (SGX), which is a set of extensions to the Intel x86 architecture that aims to reduce the level of trust required of the platform owner at runtime. In particular, SGX is designed to enable secure computation under the assumption that the whole software stack, including the operating system (OS), is malicious. In order to achieve such strong security guarantees, SGX introduces runtime environments known as *enclaves* that are isolated from the software running on the computer. To ensure isolation, SGX strictly controls the entry to and exit from enclaves and limits inadvertent state transfer from enclaves to the outer untrusted software.

In order to allow the user to distinguish between legitimate and properly configured hardware and potentially corrupted software emulators, SGX supports a *remote attestation* protocol that allows users to verify the legitimacy of the enclave before sending sensitive data to the enclave. The remote attestation protocol can verify if the enclave is running on a supported, and hence presumably trusted, processor and that the enclave has been initialized correctly. Remote attestation has been implemented through a combination of two *architectural enclaves*, the *provisioning enclave* and the *quoting enclave*, which together implement the Enhanced Privacy ID (EPID) protocol of [BL11]. EPID generates signatures that can be verified by a trusted *attestation server*. While the attestation server is trusted to verify the signatures, it is not trusted to maintain users' privacy. Consequently, SGX uses blinded group signatures which are unlinkable, allowing the attestation server to verify the signature's validity without knowing the signer's identity.

Whereas side-channel attacks have been extensively used to attack cryptographic primitives of all kind (see Chapter 6), not much is known about the effects of side-channel attacks on long term privacy of SGX enclaves. In particular, while microarchitectural side-channel attacks can be used to extract information from third-party software running within SGX enclaves [XLCZ17], the effects of side-channel attacks on the enclave's own attestation mechanism have not been properly understood. Thus, in this chapter we investigate the following questions.

> **Question 88.** *How do side-channel attacks affect SGX's EPID attestation protocol? More specifically, can side-channel attacks be used to violate EPID's forward or backward privacy?*

7.1.1   Our Contribution

A Side-Channel Attack on Intel's EPID Protocol   In this chapter, we answer the above question in the affirmative, by presenting the first cache attack on Intel's EPID protocol, as implemented inside SGX's quoting enclave. Our attack is able to recover part of the enclave's long term secret key, thereby allowing a malicious attestation server operator to break the unlinkability guarantees of SGX's remote attestation protocol.

Lattice Attacks on Zero Knowledge Proofs   As we show in Section 7.3, Intel's implementation of the EPID protocol partially leaks the length of the randomness used for one of the zero-knowledge proofs used during EPID attestation. In order to recover part of the target's long term secret key, we use this information to build an instance of the Hidden Number Problem (HNP) [BV96], which we solve using lattices. While such an approach was previously used [NS02, NS03, BT11, vSY15, GPP+16] for the case of nonce leakage from digital signatures, in this chapter we extend this approach from digital signatures to zero-knowledge protocols. To the best of our knowledge, this is the first application of side channels and lattice techniques beyond signatures, to the more general case of zero-knowledge protocols.

Attack Evaluation and Error Handling We evaluate the efficiency of our lattice attack in this setting, including measuring the effects of different optimizations, the tradeoffs involved in incorporating samples of different sizes into key recovery, and the robustness of the lattice construction against the types of measurement errors we encountered in our attack. In particular, we give experimental evidence that the lattice attack can still succeed even when a small number of erroneous traces are included, for the type of error we observed in our measurements, where the side-channel observation undercounted the true nonce length by several bits.

7.1.2 Targeted Software and Hardware

Attack Scenario In line with previous attacks on SGX [LSG⁺17, MIE17], we assume that the attacker has root access to a Linux machine. The attacker can control the OS resources, including assignment of processes to cores, interleaved execution of SGX enclave and a cache monitoring process, as well as configuring the processor power and frequency scaling. While powerful, these are valid assumptions for attacking SGX enclaves as SGX excludes the OS from its trusted computing base and assumes that the OS is malicious.

As a motivating example for the attack scenario, we look at Signal's Private Contact Discovery Service[2]. This service allows clients to probe their contact list without revealing the probe results to any of the service operators. To protect the service, Signal implements it in an SGX enclave, and employs secret, unlinkable provisioning which hides the identity of the servers providing the service from the attestation provider. However, a malicious attestation provider can use our side-channel attack in conjunction with the information it gets as part of the EPID protocol to find the host's private key. This private key can then be used to link all attestation requests for services running on the host, exposing the service to the attestation provider.

To the best of our knowledge, at the moment Intel is the only attestation provider and thus our attack currently only allows Intel to break EPID's unlinkability property. However, in principle, SGX's design also allows for (less trusted) third-party attestation providers. Unless mitigated, our attack would apply to these parties as well.

Hardware In principle, our attack is applicable to any CPU that supports Intel's SGX and EPID attestation. We empirically demonstrate our attack on a Dell Inspiron 5559 laptop with an Intel Skylake i7-6500U CPU featuring two hyper-threaded physical cores. Each physical core has 32 kB of L1D cache, used as our side channel. The L1 cache is 8-way associative and consists of 64 sets.

Software Our target laptop is running Ubuntu 16.04 and SGX Software Development Kit (SDK) version 1.7. The targeted quoting enclave and EPID library matches between the Intel prebuilt binaries and the SGX SDK source code[3].

Because the EPID signatures in the quoting enclave implementation are encrypted to a hard-coded RSA public key before being transmitted to the remote attestation provider, as described in Section 7.3, we modified the quoting enclave to encrypt to our own public key so that we could decrypt the messages. This extra encryption in the implementation is not part of the EPID attestation protocol. Our threat model for this scenario is that the remote attestation provider is the attacker.

We implemented the side-channel attack using CacheZoom[4] [MIE17]. Our signal analysis heuristics are developed using Matlab version R2017a and its signal analysis toolbox.

7.2 Preliminaries

7.2.1 Using a cache attack: Prime+Probe

The cache hierarchy of modern Intel processors consists of three levels, with each level being larger and slower than the levels above it. The L1 cache at the top of the hierarchy is split into two caches: the L1 Data (L1D) cache is used for the data the program accesses; the L1 Instruction (L1I) cache stores the instructions that the programs execute. The L1D cache is virtually indexed, *i.e.,* the processor uses the virtual address to find the cache set that stores the data. Consequently, by targeting the L1D cache in

---

[2]https://github.com/whispersystems/ContactDiscoveryService/
[3]SGX SDK is accessible at https://github.com/01org/linux-sgx
[4]CacheZoom source code is accessible at https://github.com/vernamlab/CacheZoom

our attack, we avoid the need to map the cache as was required for past works that target the next level and larger caches [LYG$^+$15, IGI$^+$16, YGL$^+$15]. The Prime+Probe methodology, explained in Chapter 6, scales well to the SGX threat model where an adversary, e.g. a malicious OS, shares the same cache while being unable to access enclave memory pages.

7.2.2 Intel SGX

Intel Software Guard Extensions (SGX) are extensions of the Intel instruction set that provide trusted execution environments (TEEs) in Intel processors. The extensions, available since the Skylake processor generation [AMG$^+$15], introduce secure execution environments called *enclaves*, that only include the processor hardware in their trusted computing base (TCB). Enclaves are protected through a combination of hardware measures that encrypt the enclave memory and strictly control the processor state at entry to and exit from the enclave.

Because SGX excludes the OS from its trusted computing base, the OS is assumed to be malicious. This, combined with SGX's lack of protection against side-channel attacks [AMG$^+$15, CD16], have paved the way for stronger attack models that include adversarial control of the OS. Several side channels exploiting these adversarial powers have been demonstrated, including attacks on the page tables [XCP15, VBWK$^+$17], branch target buffers (BTB) [LSG$^+$17], caches [SWG$^+$17, MIE17, BMD$^+$17] and memory false dependency [MES18].

One notable technique that has been used across multiple attacks [LSG$^+$17, MIE17] is exploiting the operating system's power to interrupt the enclave frequently. The technique allows the adversary to get information at a high temporal resolution and monitor memory accesses of almost every single instruction performed by the victim enclave.

Mechanisms to protect SGX enclaves against side-channel attacks have also been proposed: page table attacks can be mitigated through compiler-level page table masking [SCNS16] or through minor modifications to the mechanism of page table entries (PTE) within the SGX hardware [SP17]. Other compiler-level protections have been proposed based on software diversity and binary code retrofitting to mitigate cache attacks [WWB$^+$17, BCD$^+$17, BCD$^+$19]. *Déjà Vu* aims to detect excessive interruption introduced by OS adversaries [CZRZ17]. However, none of these mitigations have been adopted by the Intel provisioning enclave and quoting enclave. Further, the soundness and efficiency of these ad-hoc solutions have not been verified in practice.

Using an untrusted OS implies that users of the enclave need some mechanism to ensure that they communicate with a legitimate enclave that has been initialized correctly. To achieve this, Intel provides support for *remote attestation*. Remote attestation relies on a combination of secret keys stored within the processors and a cryptographic protocol which allows users to verify that they communicate with an enclave (as opposed to communicating with fake software set up by a malicious OS), that the enclave is running on a supported, and hence presumed trusted, processor and that the enclave has been initialized correctly. The protocol used for remote attestation, the Enhanced Privacy ID (EPID), is described in 7.2.4.

7.2.3 Bilinear Maps

Following the notation of [BBS04], let $\mathbb{G}_1$ and $\mathbb{G}_2$ be prime order multiplicative cyclic groups with generators g_1 and g_2 (respectively). For convenience, we recall here the definition of a pairing introduced in Definition 3 in the introduction. We say that a mapping $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ is an admissible bilinear map if the following properties hold.

1. **Bilinearity** For all $(u, v) \in \mathbb{G}_1 \times \mathbb{G}_2$ and for all $a, b \in \mathbb{Z}$, it holds that $e(u^a, v^b) = e(u, v)^{ab}$.

2. **Non-Degeneracy** $e(g_1, g_2)$ is a generator of $\mathbb{G}_T$ and $e(g_1, g_2) \neq 1$.

3. **Efficiently Computable** It is possible to efficiently compute $e(u, v)$ for all $(u, v) \in \mathbb{G}_1 \times \mathbb{G}_2$.

As we focus on the EPID construction of [BL11], we now review two parameter choices for $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$. Indeed, in order to achieve 80-bit security level, [BL11] initialize $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ with a family of 170-bit non-supersingular elliptic curves defined by Miyaji et al. [MN01]. Next, for 128-bit security, [BL11] follows the suggestion of Koblitz and Menezes [KM05] and uses a 256-bit elliptic curve which has a suitable

admissible bilinear map. As only the version offering 128-bit security is implemented in Intel's SGX SDK, we focus on this version of EPID which operates over the Fp256BN curve as standardized in [Int09]. This curve has embedding degree 12. However, our techniques are also applicable to the 80-bit version of EPID.

7.2.4 Enhanced Privacy ID

7.2.4.1 Overview

Enhanced Privacy ID (EPID) is a protocol proposed to allow remote attestation of a hardware platform without compromising the device's privacy [BL11]. EPID allows a platform to sign objects without exposing the platform identity to the verifiers, and it protects against adversaries who try to link multiple signatures to the same platform. Following the notation of [BL11], an EPID scheme consists of four entities. An Issuer $\mathcal{I}$, a revocation manager $\mathcal{R}$, a platform $\mathcal{P}$ and a verifier $\mathcal{V}$. The revocation manager maintains two revocations lists Priv-RL and Sig-RL. The scheme operates as follows:

Setup First, the issuer $\mathcal{I}$ runs the scheme's setup algorithm Setup, on input 1^k, where k is the security parameter, obtaining a group public key gpk and a issuer secret key isk. That is,

$$(\mathsf{gpk}, \mathsf{isk}) \leftarrow \mathsf{Setup}(1^k).$$

Join Next, each platform $\mathcal{P}_i$ and the issuer $\mathcal{I}$ perform a Join protocol where $\mathcal{I}$'s input is $(\mathsf{gpk}, \mathsf{isk})$ and $\mathcal{P}_i$'s input is gpk. The Join protocol terminates with $\mathcal{P}_i$ learning a secret key sk_i which it will use to sign messages to the verifier $\mathcal{V}$. More formally,

$$\langle \bot, \mathsf{sk_i} \rangle \leftarrow \mathsf{Join}_{\mathcal{I}, \mathcal{P}_i} \langle (\mathsf{gpk}, \mathsf{isk}), \mathsf{gpk} \rangle.$$

Sign In order to sign a message m, using a group public key gpk, a secret key $\mathsf{sk_i}$, and a signature based revocation list Sig-RL, the ith platform $\mathcal{P}_i$ runs the signing algorithm Sign. If $\mathsf{sk_i}$ is not revoked (i.e., $\mathsf{sk_i} \notin$ Sig-RL), Sign outputs a signature σ. Otherwise, Sign outputs $\bot$. Formally,

$$\bot/\sigma \leftarrow \mathsf{Sign}(\mathsf{gpk}, \mathsf{sk}_i, m, \mathsf{Sig\text{-}RL}).$$

Verify To verify a signature σ on a message m using a group public key gpk, a private key-based revocation list Priv-RL and a signature-based revocation list Sig-RL, the verifier $\mathcal{V}$ executes the verification algorithm Verify. This algorithm outputs valid or invalid. The invalid output indicates that either σ is not a valid signature on the message m or that the platform P_i signing m has been revoked. Formally,

$$\mathsf{valid/invalid} \leftarrow \mathsf{Verify}(\mathsf{gpk}, m, \mathsf{Priv\text{-}RL}, \mathsf{Sig\text{-}RL}, \sigma).$$

Revoke EPID supports two types of revocations. In case the revocation manager $\mathcal{R}$ knows the private key of the platform $\mathcal{P}_i$ it wishes to revoke, it simply adds sk_i to the private key-based revocation list Priv-RL by running the revoke algorithm. This results in an updated private key-based revocation list. Formally, $\mathcal{R}$ performs

$$\mathsf{Priv\text{-}RL} \leftarrow \mathsf{Revoke}(\mathsf{gpk}, \mathsf{Priv\text{-}RL}, \mathsf{sk}_i).$$

Moreover, if $\mathcal{R}$ wants to revoke some platform based on a message-signature pair (m, σ) it signed but without knowing its secret key, he can also execute the Revoke algorithm using $\mathsf{gpk}, \mathsf{Priv\text{-}RL}, \mathsf{Sig\text{-}RL}, m, \sigma$ as inputs. This results in an updated signature-based revocation list Sig-RL. Formally,

$$\mathsf{Sig\text{-}RL} \leftarrow \mathsf{Revoke}(\mathsf{gpk}, \mathsf{Priv\text{-}RL}, \mathsf{Sig\text{-}RL}, m, \sigma).$$

7.2.4.2 Security Properties of EPID

In this section we briefly overview the security properties of the EPID scheme of [BL11]. We refer the reader to [BL11] for formal discussions and definitions.

Correctness Informally, the correctness requirement states that every signature σ generated by a platform $\mathcal{P}_i$ on a message m with a secret key sk_i is valid, unless $\mathcal{P}_i$ has been revoked. More formally, let Σ_i be the set of all signatures generated by $\mathcal{P}_i$. We require that $\mathsf{Verify}(\mathsf{gpk}, m, \mathsf{Priv\text{-}RL}, \mathsf{Sig\text{-}RL}, \mathsf{Sign}(\mathsf{gpk}, \mathsf{sk}_i, m, \mathsf{Sig\text{-}RL})) = $ valid if and only if $\mathsf{sk}_i \notin$ Priv-RL and $\Sigma_i \cap \mathsf{Sig\text{-}RL} = \emptyset$.

Unlinkability At a high level, EPID's unlinkability requirement guarantees that it is impossible to identify the platform that produced a signature σ on some message m, nor is it possible to identify other signatures signed by the same platform. More specifically, even in case a malicious issuer colludes with a malicious verifier (such as in the case of a malicious remote attestation provider), knowing gpk, isk and a list of message signature pairs $(m_i, \sigma_i)_{i=1,\cdots,n}$ is not sufficient for linking a pair m, σ to a specific secret key sk or to other signatures signed by the same secret key.

Unforgeability At a high level, EPID's unforgeability requirement states that it is impossible for the attacker to forge a valid signature on some previously-unsigned message, without knowing a non-revoked secret key. This holds even in the case when the attacker knows previously revoked secret keys, belonging to compromised platforms. We refer the reader to [BL11] for a more formal discussion.

7.2.4.3 The Signing Algorithm

In this work, we attack the signing algorithm used by $\mathcal{P}$ to sign a message m. We will give a high-level description of EPID's signing algorithm in order to motivate the attack. We refer the reader to [BL11] for the full details.

Let p be the order of the bilinear group pair $(\mathbb{G}_1, \mathbb{G}_2)$ and let $e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T$ be an admissible bilinear pairing (as defined in 7.2.3). The secret key sk_i used by the ith platform $\mathcal{P}_i$ to sign m consists of $\mathsf{sk}_i = (A, x, y, f)$, where f is a random element of $\mathbb{Z}_p$ and (A, x, y) is a BBS+ signature [ASM06] on f. Following [BL11] and [CS97], we use the notation $SPK\{(a) : y = z^a\}(m)$ to denote a signature on a message m where the signature scheme used to sign m is derived from some interactive zero knowledge proof-of-knowledge protocol via the Fiat-Shamir heuristic. In this notation, the parenthesized values (a) are known to the platform $\mathcal{P}_i$ but not to the verifier $\mathcal{V}$ while all other values are public and thus known to both $\mathcal{P}_i$ and $\mathcal{V}$. Let Sig-RL be a signature-based revocation list, at a high level the algorithm proceeds as follows:

1. Choose randomly $B \leftarrow \mathbb{G}_3$ and compute $K := B^f$, where $\mathbb{G}_3$ is a cyclic group of order p defined as part of the scheme group public key gpk.

2. Choose a random $a \leftarrow \mathbb{Z}_p$, and compute $b \leftarrow y + ax$, and $T \leftarrow A \cdot h_2^a$ where $h_2 \in \mathbb{G}_2$ is also part of the scheme group public key gpk.

3. Run the following signature of knowledge protocol

$$SPK\{(x, f, a, b) : B^f = K \ \wedge \ e(T, g_2)^{-x} \cdot e(h_1, g_2)^{r_f} \cdot e(h_2, w)^{r_a} = e(T, w)/e(g_1, g_2)\}(m)$$

   where

   (a) $r_x \leftarrow \mathbb{Z}_p, r_f \leftarrow \mathbb{Z}_p, r_a \leftarrow \mathbb{Z}_p$, and $r_b \leftarrow \mathbb{Z}_p$ are chosen uniformly at random.

   (b) $c \leftarrow H(\mathsf{gpk}, B, K, T, R_1, R_2, m)$, where H is a cryptographic hash function.

   (c) $s_x \leftarrow r_x + cs$, $s_f \leftarrow r_f + cf$, $s_a \leftarrow r_a + ca$, and $s_b \leftarrow r_b + cb$ where all computations are performed over $\mathbb{Z}_p$.

   (d) The values g_1, g_2, h_1, h_2 are specified in the group public key gpk.

4. Set $\sigma_0 = (B, K, T, c, s_x, s_f, s_a, s_b)$.

5. Let Sig-RL $= \{(B_1, K_1), \ldots, (B_{n_2}, K_{n_2})\}$. For all i, compute $\sigma_i = SPK\{(f) : K = B^f \wedge K_i \neq B_i^f\}(m)$.

6. If any zero-knowledge proofs in Step 4 fails, then output $\sigma = \bot$. Otherwise, output the signature $\sigma = (\sigma_0, \sigma_1, \cdots, \sigma_{n_2})$.

Notice that if the attacker is able to leak the value of f by mounting a side-channel attack on the exponentiation routine, he is able to break the unlinkability property of the EPID construction by linking $\mathcal{P}$ to all its signatures, including past and future signatures. Next, notice that in Step 3 of the above description, the platform $\mathcal{P}$ generates a random secret nonce $r_f \in \mathbb{Z}_p$ and computes an exponentiation $e(h_1, g_2)^{r_f}$ (where $h_1 \in \mathbb{G}_1$ and $g_2 \in \mathbb{G}_2$ were both published in gpk). The signature component σ_0

includes the value $s_f = r_f + cf$ where c is the hash over several public values generated during the signing process.

As we show in Section 7.5, by recovering additional side-channel information about the length of the nonce r_f from several such signature operations, we are able to mount a lattice attack on the EPID construction and completely recover the value of f. Since the values of B and K are also part of σ, we are able to break the unlinkability property of the EPID construction by checking whether B^f equals K.

7.3 SGX EPID Provisioning and Attestation

Intel implements the EPID attestation through a combination of two enclaves, collectively known as the architectural enclaves, and deployed as part of the SGX SDK. The *provisioning enclave* is responsible for performing the EPID Join operation, effectively verifying the SGX hardware to the Intel key facility and obtaining the attestation key. The *quoting enclave* implements the EPID Sign operation. It uses the previously obtained attestation key to generate the signature attestation on each request. Note that the provision operation only needs to be performed once when a trusted environment is being prepared.

7.3.1 Provisioning and Quoting Enclave Implementations

The provisioning enclave is signed by Intel and has a special attribute that allows it to derive the permanent provision key from the hardware fused provision secret. Intel SGX supports multiple keys for different operations that can be retrieved using a special CPU instruction. Two of these keys, the provision key and the provision seal key, are only accessible to the provisioning enclave. The provisioning enclave uses an ECDSA operation and the provision key to sign a message that authenticates the SGX hardware to the Intel provisioning servers. However, the signing operation does not directly use the 128-bit provision key. Instead, a key-wrapping operation using AES-CMAC [SPLI06] over a hardcoded plaintext generates the 256-bit ECDSA key. We skip the explanation of the entire authentication protocol between the provisioning enclave and Intel provisioning server. After the authentication, the provisioning enclave obtains the private attestation key f. The provisioning enclave uses the provision seal key to store f on the disk in an encrypted form.

The quoting enclave unseals the attestation key stored by the provisioning enclave on each attestation request. The attestation process follows the EPID scheme and results in an EPID signature. However, in the quoting enclave implementation, this signature is encrypted using a hybrid RSA-AES-CMAC based on a hardcoded RSA public key. While this operation is not part of the EPID scheme it does add a layer of security to the EPID signatures used in SGX.

From an attacker perspective, this encryption hides the otherwise public EPID signature. Consequently, to be able to mount the attack we describe below, the attacker needs to be able to decrypt the EPID signatures. To allow us to decrypt the signatures, we modify the quoting enclave to use our own public key, for which we know the private key. We then sign the modified enclave. In order to avoid potential changes due to differences in the build environment, the change is applied to the binary file, rather than to the source. Hence, aside from the RSA public key and the enclave metadata, our quoting enclave is identical to the original.

We note that to perform the EPID verification, the attestation provider must be able to decrypt the EPID signatures. Hence, a real attack is only possible from a malicious attestation provider. Nevertheless, one of the main aims of the EPID protocol is to protect the privacy of the users and prevent a malicious attestation provider from linking signatures to the hosts that signed them. Hence, our attack enables a malicious attestation provider to break one of the main objectives of using the EPID protocol.

7.3.2 Scalar Multiplication in the Quoting Enclave

To perform a scalar multiplication, the quoting enclave recodes the scalar s using an extension of the Booth recoding [Boo51] introduced in Chapter 6 and uses a fixed-window algorithm with a window size of 5 and the recoded scalar. Recoding with a window size w represents the scalar as a sequence of digits s_i such that $-2^{w-1} \le s_i \le 2^{w-1}$ and $s = \sum_i 2^{wi} s_i$.

The algorithm, for which we provide the pseudo-code in Algorithm 17, precomputes the values $P_i = P^i$ for an input point P and $0 \le i \le 2^{w-1}$. It then scans the scalar from the most significant non-zero digit to the least significant digit. For each digit, it performs w squaring operations of an intermediate

---

Algorithm 17 Scalar Multiplication in the Quoting Enclave

1: **procedure** MULPOINT(point P, window size w, scalar s represented as $s_0 \ldots s_n$ using the Booth recoding)
2:      $P_0 \leftarrow \mathcal{O}$
3:      **for** $i \leftarrow 1$ **to** 2^{w-1} **do**
4:          $P_i \leftarrow P \cdot P_{i-1}$
5:      $i \leftarrow \max\{j : s_j \neq 0\}$
6:      $r \leftarrow P_{|s_i|}$
7:      $i \leftarrow i - 1$
8:      **while** $i \geq 0$ **do**
9:          $r \leftarrow r^{2^w}$
10:          $t \leftarrow P_{|s_i|}$
11:          $r \leftarrow r \cdot \text{ctSelect}(t, t^{-1}, \text{isNegative}(s_i))$
12:          $i \leftarrow i - 1$
13:      **return** r

---

result r followed by a multiply of the precomputed value matching the value of the digit or its inverse (line 12). To protect against cache attacks, the algorithm uses a constant-time select operation for the decision whether to use the precomputed value or its inverse. Furthermore, it uses the scatter-gather approach [GGO+09, BGS06] to mask the cache fingerprint of accesses to the precomputed values. For simplicity, we omit this scatter-gather operation from the algorithm.

Despite the countermeasures taken, the algorithm leaks the length of the recoded representation of the scalar. More specifically, Algorithm 17 starts from the most significant non-zero digits (line 6), leaking the length of the Booth representation of the scalar. The length of the recoded representation corresponds to the number of leading zero bits in the scalar. A full-length recoded scalar has 52 digits and the main loop of Algorithm 17 iterates 51 times. When both bits 255 and 256 of the scalar are 0, the recoded scalar has 51 digits and the loop iterates 50 times. Each additional five clear bits correspond to shortening the recoded scalar by one digit and to a resulting decrease in the number of iterations through the loop. Thus an adversary who can count the number of iterations through the main loop of Algorithm 17 or accurately measure the time it takes to perform the scalar multiplication can recover the values of some of the most significant bits of the scalar.

7.4 Short Scalar Leakage via High Resolution Side Channels

In order to extract the key leakage of EPID from an SGX quoting enclave we monitor the number of loop iterations via the L1 data cache, which is a convenient channel providing high measurement resolution.

7.4.1 Controlled Prime+Probe Attack

In this attack, we follow the scenario of [MIE17] and apply a high-resolution Prime+Probe attack in a controlled environment with respect to the OS adversarial model. More specifically,

1. The processor is configured to operate on a constant frequency to avoid dynamic changes of frequency. This reduces noise by making time measurements more accurate.

2. The thread running the quoting enclave and the Prime+Probe code is isolated on one physical core, while all other tasks of the system are placed on other physical cores of the system. This removes noise caused by memory activities of irrelevant operations from the monitored core and its L1 caches.

3. The timer interrupt handler on the attack core is configured to be triggered with a very high frequency. Thus, the quoting enclave thread can only execute a few instructions between each interrupt. In the interrupt handler, we perform Prime+Probe on the 64 L1D cache sets. As the target quoting enclave is only able to perform a few memory operations in each time frame between two consecutive interrupts, the Prime+Probe reveals all memory accesses of the enclave with high temporal and spatial resolution.
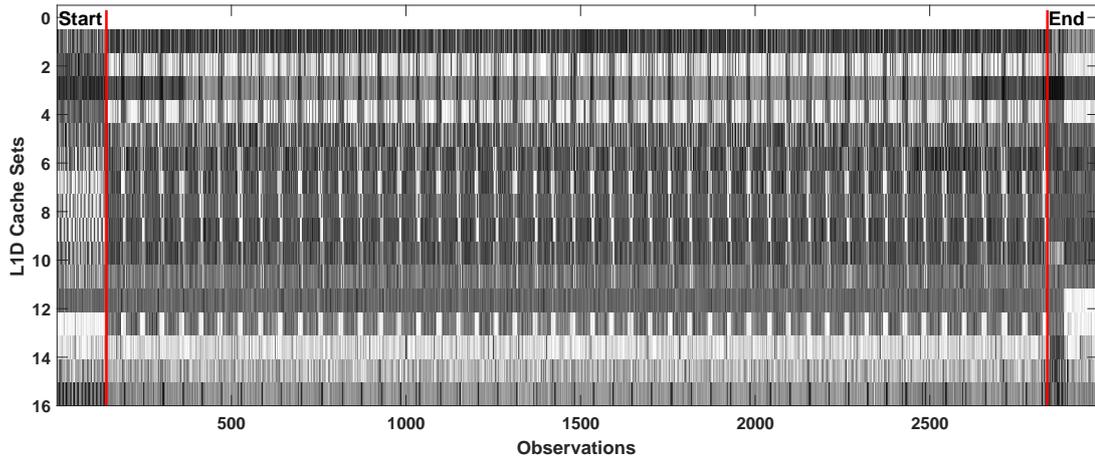
Figure 7.1: Heatmap of 16 different cache sets for the scalar multiplication. Each cache set that has a repetitive memory pattern, for example sets 7, 9 and 13, shows 48 repetitions, indicating that the ephemeral key bit size is less than $(48 + 1) \cdot 5 = 245$ bits. The red lines mark the start and the end of the repetitive memory pattern.

Figure 7.1 shows the observations of 16 different cache sets for the quoting enclave Booth multiplier. Each loop of the multiplier executes several memory operations affecting different cache sets at different times in a periodic way. As a result, we can count the number of iterations of the main loop of Algorithm 17 by looking at any cache set that has a periodic memory access pattern. Because the main loop performs one iteration for each w-bit digit following the first non-zero digit, counting the number of iteration provides information on the bit length of the scalar.
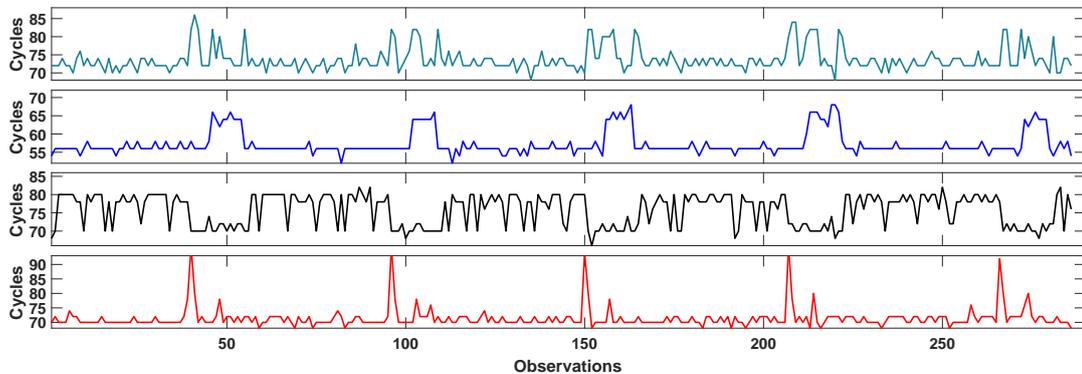
Figure 7.2: Cache access patterns on four different sets during the computation of the main loop of Algorithm 17. Each set features a different repeating pattern of the same length that can be used to count the number of loop iterations executed by the quoting enclave.

7.4.2 Loop Counting Analysis

Our goal is to determine the loop count for the Booth recoding of the scalar r_f using the above-described side-channel setup and to detect signatures that have been generated using short scalars. Our attack setup is configured to start the interrupted Prime+Probe measurement right before the call to the quoting enclave and to finish right after the enclave exits. The observations are stored in a circular buffer capable of recording 50,000 samples. The loop repetition leakage affects several cache sets in different ways, as shown in Figure 7.2.

175
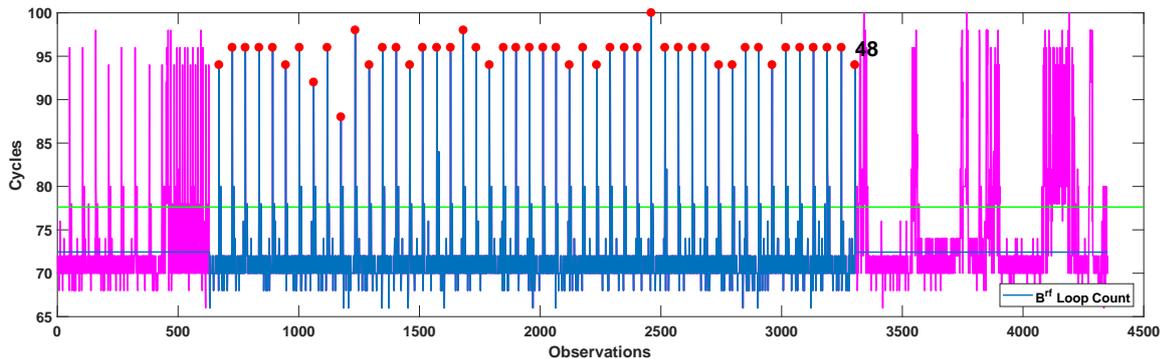
Figure 7.3: Counting loop iterations on set `02`: The number of equally spaced high peaks within a defined signal pattern reveals the number of loop iterations to be 48 in this example.

Automatically Counting Loop Iterations To automate the extraction of the number of loop iterations, we implemented several heuristics using the Matlab signal processing toolbox. A first layer locates the window for the start and end of the multiplier within the trace of 50,000 sample points. A second filter counts the number of repeated cache access patterns within the relevant window, which directly corresponds to the number of executed loops of the scalar multiplication. This information can be extracted from the periodic leakage in several of the cache sets. As depicted in Figure 7.2, the signal pattern from the main loop of Algorithm 17 is unique for each cache set. We use five different loop counters that use the information from four cache sets to count the number of loops on each signature. The first four counters detect periodic behavior to each of the four cache sets separately, while the last counter performs a reverse check on set `02`. Figure 7.3 shows a successful loop counting on set `02` that returns a count of 48 iterations, thus revealing the 12 most significant bits of the scalar are clear.

Handling Measurement Noise Even though the L1D cache channel has a very high resolution, there is still some noise that can result in a failure to count the correct number of loop iterations for each of the five counters. Common sources of error are failing to accurately detect the beginning and the end of the multiplier window, under-counting short peaks and over-counting occasional noises that introduce unexpected peaks or pattern within the sampling of a multiplier window. However, our experimental results show that if four of the five loop counters agree on the number of loop iterations, the loop counting would be error free. We automatically analyzed 11,080 signature operations and found 1214 *50-loop*, 39 *49-loop* and 2 *48-loop* short keys without any error.

Further Reducing the Number of Signatures via Manual Processing Although the fully automated approach returns enough short-key signatures to recover the key, it discards many samples that carry valuable information. Scalars resulting in a *49-loop* occur with a probability of 1/128. Thus, the automated approach, while ensuring no false positives, only detects about 45% of such occurrences. The number of required signature observations can be reduced by combining the automatic loop counting with manual verification. By returning all traces where three or more counters agree, the automatic loop counting returns 1723 *50-loop*, 54 *49-loop* and 5 *48-loop* candidates, with 0.4%, 5.0% and 0.0% error, respectively. Further reducing to the minimum of two matching counters yields 2155 *50-loop*, 117 *49-loop* and 7 *48-loop* candidates, with 5.7%, 52% and 14% error, respectively. In this case, manual post-processing is necessary, but certainly tractable, e.g. for the 117 *49-loop* candidates. We manually verified 59 of the *49-loop* candidates, thus increasing the ratio of found occurrences of *49-loop* candidates to 68%.

7.5 A Lattice Attack on EPID

The side channel gives us information about the length of r_f used to compute the signature component $s_f = r_f + cf \pmod{p}$, where s_f and c are public information, p is the 256-bit order of the elliptic curve, and f is the platform's secret membership key. We will use this information to solve for f. The problem we consider fits into the original Hidden Number Problem setting explained in Chapter 6. We begin by

176

explaining how to convert our problem to a Hidden Number Problem instance.

7.5.1 Conversion to a Hidden Number Problem

In the following, we will drop the subscript f from the signature component, and use a subscript i to index the sample number. We obtain many samples $\{(s_i, c_i)\}_i$ satisfying

$$s_i \equiv r_i + c_i f \pmod{p}, \tag{7.1}$$

as well as information about whether the number of most significant zero bits in r_i is 0, 2, 7, or 12. That is, we learn that $|s_i - c_i f| < p/2^{l_i}$ where l_i is the number of 0 bits we learn from r_i. This is identical to the Hidden Number Problem as described in Chapter 6.

Rebalancing the Nonces. Recall that in Chapter 6, Section 6.3.2.1 we mentioned how to renormalize the signatures to obtain an equivalent problem which in practice requires less samples. This optimization can be applied here.

Naively, the number of bits of r_i that we learn is 2 bits for a *50-loop* sample, 7 bits for a *49-loop* sample, and 12 bits for a *48-loop* sample. However, these r_i values are positive, while the lattice construction works with both positive or negative values for r_i. Since we know the length of the r_i, $r_i \leq 2^{n_i}$, where $n_i = 256 - l_i$ in our application, we can recenter the r_i around 0 to reduce the size of our solution by one bit. That is, we can rewrite Equation 7.1 as $s_i' - r_i' \equiv c_i f \bmod p$, with $s_i' = s_i - 2^{n_i}$ and $r_i' = r_i - 2^{n_i}$. In this way, we obtain a new problem with $-p/2^{l_i+1} \leq r_i \leq p/2^{l_i+1}$. The effect of rebalancing the nonces on the success probability of the attack can be seen in Figure 7.4.

7.5.2 Solving the Hidden Number Problem

As already discussed in Chapter 6, there are two standard approaches to solving the Hidden Number Problem: via lattices or via Fourier analysis [IEE00, DHMP13]. We use the lattice-based approach in this work, since it is quite efficient in the case of relatively large numbers of bits known (2, 7, or 12 in our attacks) and relatively few samples.

Solving the Hidden Number Problem via a lattice-based approach is done by solving SVP in an adequately constructed lattice. In our context, this lattice is generated by the following basis

$$B = \begin{pmatrix} 2^{l_1+1}p & \cdots & & & 0 & 0 \\ 0 & 2^{l_2+1}p & & & 0 & \vdots \\ & & \ddots & & & \\ & & & 2^{l_m+1}p & 0 & \\ 2^{l_1+1}c_1 & 2^{l_2+1}c_2 & \cdots & 2^{l_m+1}c_m & 1 & 0 \\ 2^{l_1+1}s_1 & 2^{l_2+1}s_2 & \cdots & 2^{l_m+1}s_m & 0 & p \end{pmatrix}, \tag{7.2}$$

where the re-scaling of the basis comes from the bounds $|r_i| < p/2^{l_i+1}$ after rebalancing the nonces, different for each line as the information we learn for each sample varies.

> **Question 89.** *Why do we expect to find the secret key?*

We briefly recall the argument that allows the Hidden Number Problem to recover the secret key, applied to our specific context. Because $|r_i| = |s_i - c_i f| \leq p/2^{l_i+1}$, we know that a vector containing the coefficients s_i will be particularly close to a vector containing the secret key f. Let B' be the basis formed by the upper quadrant of B, delimited by the $--$. One could then find f by solving CVP (or more precisely BDD since we know a (small) bound on the distance) in $\mathcal{L}(B')$ with target vector $(2^{l_1+1}s_1, 2^{l_2+1}s_2, \cdots, 2^{l_m+1}s_m, 0)$.

Instead, we will solve uSVP in the lattice $\mathcal{L}(B)$, constructed by adding the aforementioned target vector to the basis B'. This embedding should remind the reader of Kannan's embedding technique presented in Chapter 2. By solving uSVP in $\mathcal{L}(B)$, we hope to find

$$\mathbf{v} = (2^{l_1+1}r_1, \cdots, 2^{l_m+1}r_m, f, -p).$$
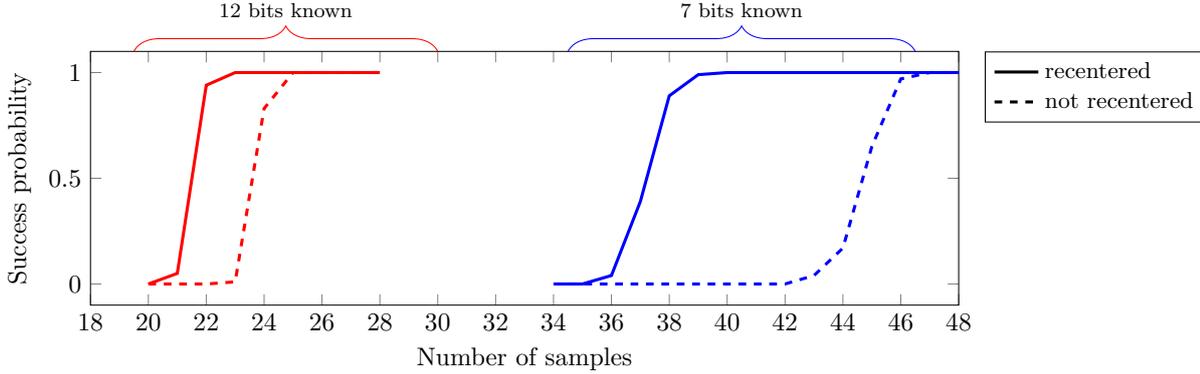
Figure 7.4: We show the success probability of key recovery using *49-loop* samples, which reveal the 7 most significant bits of the nonces, and *48-loop* samples, which reveal the 12 most significant bits learned, for different numbers of samples. Recentering the nonces has a noticeable impact on the number of samples required for key recovery.

as a short vector in the lattice. Indeed, $\mathbf{v} = \mathbf{z}B$ for some $\mathbf{z} = (z_1, \ldots, z_m, f, -1)$ with $z_i \in \mathbb{Z}$.

Benger et al. [BvSY14] noted that the shortest vector in this lattice is actually $(0, \ldots, 0, p, 0)$ and the desired vector $\mathbf{v}$ is usually the second shortest vector. We can use LLL or BKZ to compute a reduced basis. We have $\det \mathcal{L}(B) = 2^{m + \sum_i l_i} \cdot p^{m+1}$ and $||\mathbf{v}||_2 \approx \sqrt{m + 2}p$, and we hope to find $\mathbf{v}$ when $||\mathbf{v}||_2 \leq (1 + \epsilon)^{m+1}(\det B)^{1/(m+2)}$ where $1 + \epsilon$ is the approximation factor achieved by the lattice basis reduction algorithm we use.

Figure 7.4 shows the experimental success probability of the attack for *49-loop* samples, where 7 most significant bits of the nonce are known, and *48-loop* samples, where 12 most significant bits of the nonce are known.

7.5.3 Performance Tradeoffs

7.5.3.1 Using Samples of Different Lengths

Benger et al. [BvSY14] describe how to take advantage of different length samples by using different values for each l_i in the lattice basis given in Equation 7.2. Similarly, for our attack, this allows us to reduce the total number of samples we need to collect by using samples with different loop lengths. This results in a performance tradeoff: we can decrease the signature sampling time at the cost of increasing the time spent running lattice basis reduction to recover the secret key.
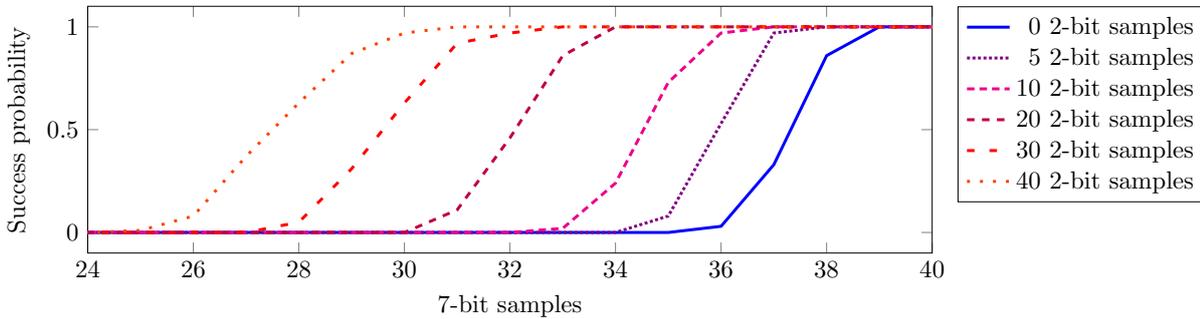
Figure 7.5: We can reduce the total number of signatures needed to carry out a successful attack by including samples of different sized nonce observations in the lattice. We constructed different sized lattices by including both samples that had revealed 7 bits of the nonce and samples that had revealed 2 bits of the nonce. The lattice dimension for each experiment is the total number of samples + 2. We measured the success probability over 100 trials on random problem instances, solved using BKZ-30.
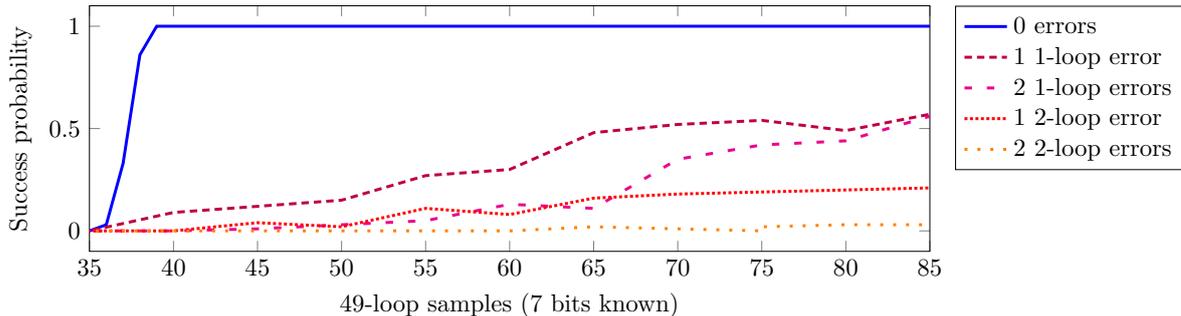
Figure 7.6: The lattice still finds the correct solution some fraction of the time, even in the presence of collection errors for the error model in our attack. We show the probability of successful recovery when the lattice contains a number of randomly generated samples whose actual nonce sizes are one or two windows, or 5 bits or 7 bits larger than the bound given in the lattice. Each point represents 100 randomly generated trials of random problem instances, solved using BKZ-30.

Using only *49-loop* samples in the lattice, corresponding to learning 7 most significant bits of the nonce, we needed 38 samples to achieve above a 50% success rate in the lattice construction. When we added 30 *50-loop* samples (in which we learn 2 most significant bits) to the lattice, we had above a 50% success rate with 30 *49-loop* samples. We show the improvement as the number of samples increases in Figure 7.5. Each point represents 100 trials using BKZ with block size 30. For the 73-dimensional lattices containing 31 *49-loop* samples and 40 *50-loop* samples (corresponding to a 100% success rate) the BKZ algorithm took 52 seconds on average to complete on a single core of our test machine.

7.5.3.2 Error Correction

It is quite common in a side-channel attack that there is some error during the collection process. In the case of our attack, an error while counting the number of loops during the modular exponentiation would result in an incorrect bound on the size of one or more of the r_i being collected. If the loop count is higher than the actual value, this is not a problem: either the sample would be excluded from the key recovery process, or the size of the r_i is still correct for the bound we use in the lattice and we expect to still find the correct solution. However, if the error happens in the other direction and the measurement process undercounts the number of loops, we will incorporate samples into the lattice whose nonce sizes are larger than the bounds we assign for them. In this case, the lattice construction can fail, because the vector may no longer correspond to the correct key.

Much of the prior work on side-channel attacks with errors for similar constructions either ignored this issue entirely, dealt with it via signal processing, or subsampled different subsets of samples until an error-free sample is obtained [GPP+16, ARAM17].

However, we find experimentally that when the lattice includes more samples than necessary, key recovery may still be possible in the presence of the types of errors we encountered in our attack. In our measurements, an error corresponds to an incorrect loop count. To model this in experiments, we generated instances of *49-loop* samples, and inserted errors corresponding to samples that should have been measured as *50-loop* or *51-loop* samples. Recall that the nonce in a *49-loop* sample is contained in the interval $[2^{244}, 2^{249})$; we generated *50-loop* errors uniformly from the interval $[2^{249}, 2^{254})$ and *51-loop* errors uniformly from the interval $[2^{254}, p)$, inserted these into our samples, and attempted key recovery. Figure 7.6 shows the success probability of our lattice construction when the lattice contains different numbers of errors. Each plotted sample represents 100 randomly generated trials with the specified type of error run with BKZ-30. The recovery rate with errors whose most significant bits are 1 is similar to the error rates for the *51-loop* samples.

As a concrete example, a 75-sample lattice with 2 1-loop errors succeeded 42% of the time in our experiments. This corresponds to a 2.9% error rate. We would expect a 39-sample lattice (which succeeded with 100% probability in our experiments) to achieve 0% errors with probability $0.97^{39} = 0.35$, or 35% of experiments.

7.5.4 Recovering f

Using the automatically classified data from the side-channel attack described in Section 7.4, we were able to recover f using BKZ-30 on 37 error-free *49-loop* samples, corresponding to 7 most significant bits of each ephemeral r_i known. It took 10,600 total signature samples to collect this data. Next, by using samples of different loop lengths we can further reduce the total number of signatures required. More specifically, in Table 7.1 we show some different strategies for successful key recovery given our empirical signature data. For the row corresponding to n signature data points, we used all of the *48-loop* (corresponding to 12 bits known) and *49-loop* (corresponding to 7 bits known) samples that were detected during the first n signature measurements in our data, and then added *50-loop* (corresponding to 2 bits known) samples from the first n signatures to the lattice until key recovery succeeded.

As described in Section 7.4.2, fewer signatures are required if manual inspection is used to help classify the signals. In that case, we would only need less than 7,500 observed signatures to obtain enough *49-loop* observations for a full key recovery.

Signatures	48-loop	49-loop	50-loop	BKZ block size	BKZ time
10300	2	35	0	2	0.1s
10000	2	31	10	20	0.2s
9000	2	29	21	30	1.4s
8000	2	25	35	30	4.5s

Table 7.1: Strategies for key recovery with different numbers of signature samples.

7.6 Conclusions

In this work, we show yet another leakage in highly sensitive code—the implementation of the EPID protocol for SGX remote attestation. While the attack only allows a malicious attestation provider to break the link signatures to a host, the unlinkability guarantee it breaks is the main reason for using the EPID protocol in the first place. From the structure of the code, it is clear that the developers have attempted to eliminate side-channel leaks. Thus, this incident demonstrates that producing constant-time code is not trivial and that better tools for facilitating such development are required.

This work extends the art of recovering the long-term key from partial information on the ephemeral keys. First, we apply known techniques used in the context of digital signatures to the wider context of zero-knowledge proofs. Second, we investigate the handling of erroneous inputs for the hidden number problem. We show that prior common belief notwithstanding, lattices can handle some erroneous input. Exploring the trade-offs between past approaches of selecting a random subset of the inputs and the new approach of using the inputs in the lattice is left for future work.

Chapter 8

Attacking ECDSA signature protocol with EHNP

Attacking ECDSA with wNAF implementation for the scalar multiplication first requires some side-channel analysis to collect information, then lattice based methods to recover the secret key. In this chapter, we reinvestigate the construction of the lattice used in one of these methods, the *Extended Hidden Number Problem* (EHNP). We find the secret key with only 3 signatures, thus reaching a known theoretical bound, whereas best previous methods required at least 4 signatures in practice. Given a specific leakage model, our attack is more efficient than previous attacks, and for most cases, has better probability of success. To obtain such results, we perform a detailed analysis of the parameters used in the attack and introduce a preprocessing method which reduces by a factor up to 7 the total time to recover the secret key for some parameters. We perform an error resilience analysis which has never been done before in the setup of EHNP. Our construction finds the secret key with a small amount of erroneous traces, up to 2% of false digits, and 4% with a specific type of error.

This chapter is joint work with Rémi Piau and Cécile Pierrot and was published in the proceedings of the Africacrypt 2020 conference [DPP20].

Contents

8.1 Introduction

The Elliptic Curve Digital Signature Algorithm (ECDSA) [JMV01], first proposed in 1992 by Scott Vanstone [Van92], is a standard public key signature protocol widely deployed. Its signing algorithm

requires scalar multiplications of a point P on an elliptic curve by an ephemeral key k. Since this operation is time-consuming and often the most time-consuming part of the protocol, it is necessary to use an efficient algorithm. The Non Adjacent Form (NAF) and its windowed variant (wNAF), presented in Chapter 6, were introduced as an alternative to the binary representation of the nonce k to reduce the execution time of the scalar multiplication. Indeed, we recall that the NAF representation does not allow two non-zero digits to be consecutive, thus reducing the Hamming weight of the representation of the scalar. This improves on the time of execution as the latter is dependent on the number of non-zero digits. The wNAF representation is present in implementations such as in Bitcoin, as well as in the libraries Cryptlib, BouncyCastle and Apple's Common-Crypto. Moreover, until recently (May 2019), wNAF was present in OpenSSL.

However, implementing the scalar multiplication using wNAF representation and no added layer of security makes the protocol vulnerable to side-channel attacks. In our case, the leakage recovered from a side-channel attack corresponds to differences in the execution time of a part of the signing algorithm, observable by monitoring the cache.

For ECDSA, cache side-channel attacks such as Flush+Reload [YB14, YF14] have been used to recover information about either the sequence of operations used to execute the scalar multiplication, or for example in [GB17] the modular inversion. For the scalar multiplication, these operations are either a multiplication or an addition depending on the bits of k. This information is usually referred to as a double-and-add chain or the trace of k. A trace is created when a signature is produced by ECDSA and thus we talk about signatures and traces in an equivalent sense. At this point, we ask how many traces need to be collected to successfully recover the secret key. Indeed, from an attacker's perspective, the least traces necessary, the more efficient the attack is. This quantity depends on how much information can be extracted from a single trace and how combining information of multiple traces is used to recover the key. We work on the latter to minimize the number of traces needed.

The nature of the information obtained from the side-channel attack allows to determine what kind of method should be carried out to recover the secret key. As already explained in Chapter 6, the Hidden Number Problem is an example of algorithm which allows to recover the secret key of ECDSA in polynomial time when consecutive bits of the nonces k are known. The basic structure of the attack algorithm is to construct a lattice which contains the knowledge of consecutive bits of the ephemeral keys, and by solving SVP, to recover the secret key. However, these results considered perfect traces, but obtaining traces without any misreadings is very rare. In 2018, Dall et al. [DDME+18] included an error-resilience analyze to their attack: they showed that key recovery with HNP is still possible even in the presence of erroneous traces. This work is described in Chapter 7.

In 2016, Fan, Wang and Cheng [FWC16] used another lattice-based method to attack ECDSA: the Extended Hidden Number Problem (EHNP) [HR07]. EHNP mostly differs from HNP by the nature of the information given as input. Indeed, the information required to construct an instance of EHNP is not sequences of consecutive bits, but the positions of the non-zero coefficients in any representation of some integers. This model, which we consider in this article as well, is relevant when describing information coming from Flush+Reload or Prime+Probe attacks for example, the latter giving a more generic scenario with no shared data with the spy. In [FWC16], the authors are able to extract 105.8 bits of information per signature on average, and thus require in theory only 3 signatures to recover a 256-bit secret key. In practice, they were able to recover the secret key using 4 error-free traces.

In order to optimize an attack on ECDSA various aspects should be considered. By minimizing the number of signatures required in the lattice construction, one minimizes the number of traces needed to be collected during the side-channel attack. Moreover, reducing the time of the lattice part of the attack, and improving the probability of success of key recovery allows to reduce the overall time of the attack. In this work, we improve on all three of these aspects. Furthermore, we propose the first error-resilience analysis for EHNP and show that key recovery is still possible with erroneous traces too.

8.1.1 Our contribution

In this work, we reinvestigate the attack against ECDSA with wNAF representation for the scalar multiplication using EHNP. We focus on the lattice part of the attack, *i.e.*, the exploitation of the information gathered by the side-channel attack. We first assume we obtain a set of error-free traces from a side-channel analysis. We preselect some of these traces to optimize the attack. The main idea of the lattice

| Number of | Our attack | | [FWC16] | |
signatures	Time	Success (%)	Time	Success (%)
3	39 hours	0.2%	–	–
4	1 hour 17 minutes	0.5%	41 minutes	1.5%
5	8 minutes 20 seconds	6.5%	18 minutes	1%
6	≈ 5 minutes	25%	18 minutes	22%
7	≈ 3 minutes	17.5%	34 minutes	24%
8	≈ 2 minutes	29%	–	–

Table 8.1: Comparing attack times with [FWC16], for 5000 experiments.

part is then to use the ECDSA equation and the knowledge gained from the selected traces to construct a set of modular equations which include the secret key as an unknown. These modular equations are then incorporated into a lattice basis similar to the one given in [FWC16], and a short vector in it will contain the necessary information to reconstruct the secret key. We call experiment one run of this algorithm. An experiment succeeds if the algorithm recovers the secret key.

A new preprocessing method. The idea of selecting good traces beforehand has already been explored in [WF17]. The authors suggest three rules to select traces that improve the attack on the lattice part. Given a certain (large) amount of traces available, the lattice is usually built with a much smaller subset of these traces. Trying to identify beforehand the traces that would result in a better attack is a clever option. The aim of our new preprocessing - that completely differs from [WF17] - is to regulate the size of the coefficients in the lattice, and this results in a better lattice reduction time. For instance, with 3 signatures, we were able to reduce the total time of the attack by a factor of 7.

Analyzing the attack. Several parameters occur while building and reducing the lattice. We analyze the performance of the attack with respect to these parameters and present the best parameters that optimize either the total time or the probability of success.

First, we focus on the attack time. Note that when talking about the overall time of the attack, we consider the average time of a single experiment multiplied by the number of trials necessary to recover the secret key. We compare our times with the numbers reported in [FWC16, Table 3] with method C. Indeed, methods A and B in [FWC16] use extra information that comes from choices in the implementation which we choose to ignore as we want our analysis to remain as general as possible. Moreover, in order to have a fair comparison with our methodology, the times reported in [FWC16] to which we compare ourselves have to be multiplied by the number of trials necessary for their attack succeed, thus increasing their total time by a lot. Using 5 signatures, their best total time would be around 15 hours instead of 18 minutes. The comparison is justified as we consider the same leakage model, and compare timings when running experiments on similar machines. For 4 signatures, our attack is slightly slower. For 4 signatures, no times are reported without method A. Thus, we have no other choice than to compare our times with theirs, using A. Yet their time for 4 signatures without A should at least be the time they report with it. However, when considering more than 4 signatures, our attack is faster. We experiment up to 8 signatures to further improve our overall time. In this case, our attack runs at best in 2 minutes and 25 seconds. Timings for 8 signatures are not reported in [FWC16], and the case of 3 signatures was never reached before our work. In Table 8.1, we compare our times with the fastest times reported by [FWC16]. We choose their fastest times but concerning our results we choose to report experiments which are faster (not the fastest) with, if possible, better probability than theirs.

The overall time of the attack is also dependent on the success probability of key recovery. From Table 8.2, one can see that our success probability is higher than [FWC16], except for 7 signatures. They have 68% of success with their best parameters whereas we only reach 45% in this case.

For the sake of completeness, we mention that in [vSY15], the authors use HNP to recover the secret key using 13 signatures. Their success probability in this case is around 54 % and their overall time is close to 20 seconds, hence much faster. However, as their leakage model is different, we do not further mention their work.

Number of	Our attack		[FWC16]	
signatures	Success (%)	Time	Success (%)	Time
3	0.2%	39 hours	–	–
4	4%	25 hours 28 minutes	1.5%	41 minutes
5	20%	2 hours 42 minutes	4%	36 minutes
6	40%	1 hour 4 minutes	35%	1 hour 43 minutes
7	45%	2 hours 36 minutes	68%	3 hours 58 minutes
8	45%	5 hours 2 minutes	–	–

Table 8.2: Comparing success probability with [FWC16], for 5000 experiments.

Finding the key with only three signatures. Overall, combining a new preprocessing method, a modified lattice construction and a careful choice of parameters allows us to mount an attack which works in practice with only 3 signatures. However, the probability of success in this case is very low. We were able to recover the secret key only once with BKZ-35 over 5000 experiments. If we assume the probability is around 0.02%, as each trial costs 200 seconds on average, we can expect to find the secret key after 12 days using a single core. Note that this time can be greatly reduced when parallelizing the process, *i.e.*, each trial can be run on a separate core. On the other hand, if we use our preprocessing method, with 3 signatures we obtain a probability of success of 0.2% and a total time of key recovery of 39 hours, thus the factor 7 of improvement mentioned above. Despite the low probability of success, this result remains interesting nonetheless. Indeed, the authors in [FWC16] reported that in practice, the key couldn't be recovered using less than 4 signatures and we improve on their result.

Resilience to errors. We also investigate the resilience to errors of our attack. Such an analysis has not yet been done in the setup of EHNP. It is important to underline that collecting traces without any errors using any side-channel attack is very hard. Previous works used perfect traces to mount the lattice attack. Thus, it required collecting more traces. As pointed out in [FWC16], more or less twice as many signatures are needed if errors are considered. In practice, this led [FWC16] to gather on average 8 signatures to be able to find the key with 4 perfect traces. We experimentally show that we are still able to recover the secret key even in the presence of faulty traces. In particular, we find the key using only 4 faulty traces, but with a very low probability of success. As the percentage of incorrect digits in the trace grows, the probability of success decreases and thus more signatures are required to successfully recover the secret key. For instance, if 2% of the digits are wrong among all the digits of a given set of traces, it is still possible to recover the key with 6 signatures. This result is valid if errors are uniformly distributed over the digits. However, we have a better probability to recover the key if errors consist in 0-digit faulty readings, *i.e.*, 0 digits read as non-zero. In other words, the attack could work with a higher percentage of errors, around 4%, if we could ensure from the side-channel attack and some preprocessing methods that none of the non-zero digits have been flipped to 0.

8.2   Attacking ECDSA using lattices

As explained in Chapter 6, by using side-channel attacks, one can recover information about the wNAF representation of the nonce k. In particular, it allows to know the positions of the non-zero coefficients in the representation of k. However, the value of these coefficients are unknown. This information can be used in the setup of the Extended Hidden Number Problem (EHNP) to recover the secret key. For many messages m, we use ECDSA to produce signatures (r, s) and each run of the signing algorithm produces a nonce k. We assume we have the corresponding trace of the nonce, that is, the equivalent of the double-and-add chain of kG using wNAF. The goal of the attack is to recover the secret α while optimizing either the number of signatures required or the total time of the attack.

8.2.1   Using EHNP to attack ECDSA

From the ECDSA algorithm introduced in Chapter 6, Section 6.3.1.2, we know that given a message m, the algorithm outputs a signature (r, s) such that

$$\alpha r = sk - H(m) \pmod{q} \tag{8.1}$$

where q is the prime order of the group, here the rational points of an elliptic curve. The value $H(m)$ is just some hash of the message m. We consider a set of u signature pairs (r_i, s_i) with corresponding message m_i that satisfy Equation (8.1). For each signature pair, we have a nonce k. Using the wNAF representation of k introduced in Chapter 6, we write $k = \sum_{j=1}^{\ell} k_j 2^{\lambda_j}$, with $k_j \in \{\pm 1, \pm 3, \ldots, \pm(2^w - 1)\}$ and the choice of w depends on the implementation. Note that the coefficients k_j are unknown, however, the positions λ_j are supposed to be known via some side-channel leakage. It is then possible to represent the ephemeral key k as the sum of a known part, and an unknown part. As the value of k_j is odd, one can write $k_j = 2k'_j + 1$, where $-2^{w-1} \le k'_j \le 2^{w-1} - 1$. Using the same notations as in [FWC16], set $d_j = k'_j + 2^{w-1}$, where $0 \le d_j \le 2^w - 1$. In the rest of the chapter, we will denote by μ_j the window-size of d_j. Note that here, $\mu_j = w$ but this window-size will be modified later. This allows to rewrite the value of k as

$$k = \sum_{j=1}^{\ell} k_j 2^{\lambda_j} = \bar{k} + \sum_{j=1}^{\ell} d_j 2^{\lambda_j + 1}, \tag{8.2}$$

with $\bar{k} = \sum_{j=1}^{\ell} 2^{\lambda_j} - \sum_{j=1}^{\ell} 2^{\lambda_j + w}$. The expression of $\bar{k}$ represents the known part of k. By substituting k in Equation (8.2), we get a system of modular equations:

$$\alpha r_i - \sum_{j=1}^{\ell_i} 2^{\lambda_{i,j}+1} s_i d_{i,j} - (s_i \bar{k}_i - H(m_i)) \equiv 0 \pmod{q} \tag{8.3}$$

where the unknowns are α and the $d_{i,j}$. The known values are ℓ_i, which is the number of non-zero digits in k for the ith sample, $\lambda_{i,j}$, which is the position of the jth non-zero digit in k for the ith sample and $\bar{k}$ defined above. Equation (8.3) is then used as input to EHNP, following the method explained in [HR07]. The problem of finding the secret key is then reduced to solving the short vector problem in a given lattice presented in the following section.

8.2.2 Constructing the lattice

Before giving the lattice basis construction, we redefine Equation (8.3) to reduce the number of unknown variables in the system. This will allow us to construct a lattice of smaller dimension. Again, we use the same notations as in [FWC16].

Eliminating one variable. One straightforward way to reduce the lattice dimension is to eliminate a variable from the system. In this case, one can eliminate α from Equation (8.3). Let E_i denote the ith equation of the system. Then by computing $r_1 E_i - r_i E_1$, we get the following new modular equations

$$\sum_{j=1}^{\ell_1} \underbrace{(2^{\lambda_{1,j}+1} s_1 r_i)}_{:=\tau_{j,i}} d_{1,j} + \sum_{j=1}^{\ell_i} \underbrace{(-2^{\lambda_{i,j}+1} s_i r_1)}_{:=\sigma_{i,j}} d_{i,j}$$
$$- \underbrace{r_1(s_i \bar{k}_i - H(m_i)) + r_i(s_1 \bar{k}_1 - H(m_1))}_{:=\gamma_i} \equiv 0 \pmod{q}. \tag{8.4}$$

Using the same notations as in [FWC16], we define $\tau_{j,i} = 2^{\lambda_{1,j}+1} s_1 r_i$, $\sigma_{i,j} = -2^{\lambda_{i,j}+1} s_i r_1$ and $\gamma_i = r_1(s_i \bar{k}_i - H(m_i)) + r_i(s_1 \bar{k}_1 - H(m_1))$ for $2 \le i \le u$, $1 \le j \le \ell_i$. Even if α is eliminated from the equations, if we recover some $d_{i,j}$ values from a short vector in the lattice, we can recover α using any equation in the modular system (8.3). We now use Equation (8.4) to construct the lattice basis.

From a modular system to a lattice basis. Let $\mathcal{L}$ be the lattice constructed for the attack, and we have $\mathcal{L} = \mathcal{L}(B)$ where the lattice basis B is given below. Let $m = \max_{i,j} \mu_{ij}$ for $1 \le j \le \ell_i$ and $2 \le i \le u$. We set a scaling factor $\Delta \in \mathbb{N}$ to be defined later. The lattice basis is given by

$$B = \begin{matrix} Eq~(8.4), i=2 & \cdots & Eq~(8.4), i=u \end{matrix}$$

$$B = \begin{pmatrix}
\Delta 2^m q & 0 & 0 & 0 & & & & & & & \\
0 & \ddots & \vdots & & & & & & & & \\
0 & \cdots & \Delta 2^m q & 0 & & & & & & & \\
\Delta 2^m \tau_{1,2} & \cdots & \Delta 2^m \tau_{1,u} & 2^{m-\mu_{1,1}} & & & & & & & \\
\vdots & & \vdots & 0 & \ddots & & & & & & \\
\Delta 2^m \tau_{\ell_1,2} & \cdots & \Delta 2^m \tau_{\ell_1,u} & & & 2^{m-\mu_{1,\ell_1}} & & & & & \\
\Delta 2^m \sigma_{2,1} & 0 & 0 & & & & 2^{m-\mu_{2,1}} & & & & \\
\vdots & & \vdots & & & & & \ddots & & & \\
\Delta 2^m \sigma_{2,\ell_2} & & \vdots & & & & & & 2^{m-\mu_{2,\ell_2}} & & \\
0 & \ddots & 0 & \vdots & & & & & & \ddots & \\
\vdots & & \Delta 2^m \sigma_{u,1} & & & & & & & & 2^{m-\mu_{u,1}} \\
\vdots & & \vdots & & & & & & & & \ddots \\
0 & 0 & \Delta 2^m \sigma_{u,\ell_u} & 0 & & & & & & & 2^{m-\mu_{u,\ell_u}} \\
\Delta 2^m \gamma_2 & \cdots & \Delta 2^m \gamma_u & 2^{m-1} & \cdots & & & & & 2^{m-1} & 2^{m-1}
\end{pmatrix}.$$

Let $n = (u-1) + T + 1 = T + u$, with $T = \sum_{i=1}^{u} \ell_i$, be the dimension of the lattice. The $u-1$ first columns correspond to Equation 8.4 for $2 \leq i \leq u$. Each of the remaining columns, except the last one, correspond to a d_{ij}, and contains coefficients that allow to regulate the size of the d_{ij}. The determinant of $\mathcal{L}$ is given by $\det \mathcal{L} = q^{u-1} (\Delta 2^m)^{u-1} 2^{\sum_{i,j}(m-\mu_{i,j})} 2^{m-1}$.

The lattice is built such that there exists $\mathbf{w} \in \mathcal{L}$ which contains the unknowns $d_{i,j}$. To find it, we know there exist some values $t_2, t_2, \ldots, t_u$ such that if $\mathbf{v} = (t_2, \ldots, t_u, d_{1,1}, \ldots, d_{u,\ell_u}, -1)$, we get $\mathbf{w} = \mathbf{v}B$, and

$$\mathbf{w} = (0, \ldots, 0, d_{1,1} 2^{m-\mu_{1,1}} - 2^{m-1}, \ldots, d_{u,\ell_u} 2^{m-\mu_{u,\ell_u}} - 2^{m-1}, -2^{m-1}).$$

If we are able to find $\mathbf{w}$ in the lattice, then we can reconstruct the secret key α. In order to find $\mathbf{w}$, we estimate its norm and make sure $\mathbf{w}$ appears in the reduced basis. After reducing the basis, we look for vectors of the correct shape, *i.e.*, with sufficiently enough zeros at the beginning and the correct last coefficient, and attempt to recover α for each of these.

How the size of Δ affects the norms of the short vectors. In order to find the vector $\mathbf{w}$ in the lattice, we reduce B using LLL or BKZ. For $\mathbf{w}$ to appear in the reduced basis, one should at least set Δ such that

$$||\mathbf{w}||_2 \leq (1.02)^n (\det \mathcal{L})^{1/n}. \tag{8.5}$$

The vector $\mathbf{w}$ we expect to find has norm $||\mathbf{w}||_2 \leq 2^{m-1}\sqrt{T+1}$. From Equation 8.5, one can deduce the value of Δ needed to find $\mathbf{w}$ in the reduced lattice:

$$\Delta \geq \frac{(T+1)^{(T+u)/(2(u-1))} 2^{\frac{1+\sum \mu_{i,j} - (u+T)}{u-1}}}{q(1.02)^{\frac{(T+u)^2}{u-1}}} := \Delta_{th}$$

In our experiments, the average value of ℓ_i for $1 \leq i \leq u$ is $\tilde{\ell} = 26$, and thus $T = u \times \tilde{\ell}$ on average. Moreover, the average value of μ_{ij} is 7 and so on average $\sum \mu_{ij} = 7 \times u \times \tilde{\ell}$. Hence, if we compute Δ_{th} for $u = 3, \ldots, 8$, with these values, we obtain $\Delta_{th} \ll 1$, which does not help us to set this parameter. In practice, we verify that $\Delta = 1$ allows us to recover the secret key. In Section 8.4, we vary the size of Δ to see whether a slightly larger value affects the probability of success.

Too many small vectors. While running BKZ on B, we note that for some specific sets of parameters the reduced basis contains some undesired short vectors, *i.e.*, vectors that are shorter than $\mathbf{w}$. This can be explained by looking at two consecutive rows in the lattice basis given above, say the jth row and the $(j+1)^{th}$ row. For example, one can look at rows which correspond to the $\sigma_{i,j}$ values but the same argument is valid for the rows concerning the $\tau_{j,i}$. From the definitions of the σ values we have $\sigma_{i,j+1} = -2^{\lambda_{i,j+1}+1} \cdot s_i r_1 = -2^{\lambda_{i,j+1}+1} \cdot (\frac{\sigma_{i,j}}{-2^{\lambda_{i,j}+1}})$. So $\sigma_{i,j+1} = 2^{\lambda_{i,j+1} - \lambda_{i,j}} \cdot \sigma_{i,j}$. Thus the linear combination given by the $(j+1)^{th}$ row minus $2^{\lambda_{i,j+1} - \lambda_{i,j}}$ times the jth row gives a vector

$$(0, \cdots, 0, -2^{\lambda_{i,j+1} - \lambda_{i,j} + m - \mu_{i,j}}, 2^{m-\mu_{i,j+1}}, 0, \cdots, 0). \tag{8.6}$$

Algorithm 18 Merging algorithm

Input: v_λ, a table of size n with the positions of non-zero digits in the trace sorted in increasing order and $n \geq 1$, a window size w.

Output: $v_{\lambda'}$, a table of size $n' \leq n$ containing the merged λ values and table v_μ of same size n', with the values of the window size μ_i.

1: $i \leftarrow 1$
2: $v_{\lambda'} \leftarrow$ empty array
3: $v_\mu \leftarrow$ empty array
4: $v_{\lambda'}.push\_back(v_\lambda.at(0))$
5: **while** $i < n$ **do**
6:      $dist \leftarrow v_\lambda.at(i) - v_\lambda.at(i-1)$
7:      **if** $dist > w + 1$ **then**
8:          $v_\mu.push\_back(v_\lambda.at(i-1) - v_{\lambda'}.last() + w)$
9:          $v_{\lambda'}.push\_back(v_\lambda.at(i))$
10:     $i \leftarrow i + 1$
11: $v_\mu.push\_back(v_\lambda.at(n) - v_{\lambda'}.last() + w)$
12: **return** $(v_{\lambda'}, v_\mu)$

Yet, this vector is expected to have smaller norm than **w**. Some experimental observations are detailed in Section 8.4.

Differences with the lattice construction given in [FWC16]. Let B' be the lattice basis constructed in [FWC16]. Our basis B is a rescaled version of $\mathcal{B}'$ such that $B = 2^m \Delta B'$. This rescaling allows us to ensure that all the coefficients in our lattice basis are integer values. Note that [FWC16] have a value δ in their construction which corresponds to $1/\Delta$. In this work, we give a precise analysis of the value of Δ, both theoretically and experimentally in Section 8.4, which is missing in [FWC16].

8.3 Improving the lattice attack

8.3.1 Reducing the lattice dimension: the merging technique

In [FWC16], the authors present another way to further reduce the lattice dimension, which they call the merging technique. It aims at reducing the lattice dimension by reducing the number of non-zero digits of k. The lattice dimension depends on the value $T = \sum_{i=1}^u \ell_i$, and thus reducing T reduces the dimension. For the understanding of the attack, it suffices to know that after merging, we obtain some new values ℓ' corresponding to the new number of non-zero digits and λ'_j the position of these digits for $1 \leq j \leq \ell'$. After merging, one can rewrite

$$k = \bar{k} + \sum_{j=1}^{\ell'} d'_j 2^{\lambda'_j + 1},$$

where the new d'_j have a new window size which we denote μ_j, i.e., $0 \leq d'_j \leq 2^{\mu_j} - 1$.

We present our merging algorithm based on Algorithm 3 given in [FWC16]. Our algorithm modifies directly the sequence $\{\lambda_j\}_{j=1}^\ell$, whereas [FWC16] work on the double-and-add chains. This helped us avoid some implementation issues such as an index outrun present in Algorithm 3 [FWC16], line 7. To facilitate the ease of reading of (our) Algorithm 18, we work with dynamic tables. Let us first recall various known methods we use in the algorithm: $push\_back(e)$ inserts an element e at the end of the table, $at(i)$ outputs the element at index i, and $last()$ returns the last element of the table. We consider tables of integers indexed in $[0; S-1]$, where S is the size of the table.

A useful example of the merging technique is given in [FWC16]. From 3 to 8 signatures the approximate dimension of the lattices using the elimination and merging techniques are the following: $80, 110, 135, 160, 190$ and 215. Each new lattice dimension is roughly 54% of the dimension of the lattice before applying these techniques, for the same number of signatures. For instance, with 8 signatures

we would have a lattice of dimension 400 on average, far too large to be easily reduced. For the traces we consider, after merging the mean of the ℓ_i is 26, the minimum being 17 and the maximum 37 with a standard deviation of 3. One could further reduce the lattice dimension by preprocessing traces with small ℓ_i. However, the standard deviation being small, the difference in the reduction times should not be affected too much.

8.3.2 Preprocessing the traces

The two main pieces of information we can extract and use in our attack are first the number of non-zero digits in the wNAF representation of the nonce k, denoted ℓ and the weight of each non-zero digit, denoted μ_j for $1 \leq j \leq \ell$. Let $\mathcal{T}$ be the set of traces we obtained from the side-channel leakage representing the wNAF representation of the nonce k used while producing an ECDSA signature. We consider the subset

$$S_a = \{t \in \mathcal{T} \mid \max_j \mu_j \leq a, 1 \leq j \leq \ell\}.$$

We choose to preselect traces in a subset S_a for small values of a. The idea behind this preprocessing is to regulate the size of the coefficients in the lattice. Indeed, when selecting u traces for the attack, by upper-bounding $m = \max_{i,j} \mu_{i,j}$ for $2 \leq i \leq u$, we force the coefficients to remain smaller than when taking traces at random.

We work with a set $\mathcal{T}$ of 2000 traces such that for all traces $11 \leq \max_j \mu_j \leq 67$. The proportion of signatures corresponding to the different preprocessing subsets we consider in our experiments are: 2% for S_{11}, 18% for S_{15} and 44% for S_{19}. The effect of preprocessing on the total time is explained in Section 8.4.

8.4 Performance analysis

Traces from the real world. We work with the elliptic curve `secp256k1` but none of the techniques introduced here are limited to this specific elliptic curve. We consider traces from a Flush+Reload attack, executed through hyperthreading, as it can virtually recover the most amount of information. In practice, measurements done during the cache attack depend on the noise in the execution environment, the threat model and the target leaky implementation.

To the best of our knowledge, the only information we can recover are the positions of the non-zero digits. We are not able to determine the sign or the value of the digits in the wNAF representation. In [FWC16], the authors exploit the fact that the length of the binary string of k is fixed in implementations such as OpenSSL, and thus more information can be recovered by comparing this length to the length of the double-and-add chain. In particular, they were able to recover the MSB of k, and in some cases the sign of the second MSB. We do not consider this extra information as we want our analysis to remain general.

We report calculations ran on error-free traces where we evaluate the total time necessary to recover the secret key and the probability of success of the attack. Our experiments have two possible outputs: either we reconstruct the secret key α and thus consider the experiment a success, or we do not recover the secret key, and the experiment fails. In order to compute the success probability and the average time of one reduction, we run 5000 experiments for some specific sets of parameters using either Sage's default BKZ implementation [The16] or a more recent implementation of the latest sieving strategies, the General Sieve Kernel (G6K) [ADH+19]. The experiments were ran using the cluster Grid'5000 on a single core of an Intel Xeon Gold 6130. The total time is the average time of a single reduction multiplied by the number of trials necessary to recover the key. For a fixed number of signatures, we either optimize the total time or the success probability. We report numbers in Tables 8.3, 8.4 when using BKZ. [5]

Comments on G6K: We do not report the full experiments ran with G6K since using this implementation does not lead to the fastest total time of our attack: around 2 minutes using 8 signatures for BKZ and at best 5 minutes for G6K.

---

[5] In [FWC16], the authors use an Intel Core i7-3770 CPU running at 3.40GHz on a single core. In order for the time comparison to be meaningful, we ran experiments with a machine of comparable performance to estimate the timings of a single reduction. As we obtained similar timings with an older machine than used in [FWC16], the variations we find when comparing ourselves to them solely come from the lattice construction and the reduction algorithm being used rather than hardware differences.

Number of signatures	Total time	Parameters			Probability of success (%)
		BKZ	Preprocessing	Δ	
3	39 hours	35	S_{11}	$\approx 2^3$	0.2
4	1 hour 17	25	S_{15}	$\approx 2^3$	0.5
5	8 min 20	25	S_{19}	$\approx 2^3$	6.5
6	3 min 55	20	S_{all}	$\approx 2^3$	7
7	2 min 43	20	S_{all}	$\approx 2^3$	17.5
8	2 min 25	20	S_{all}	$\approx 2^3$	29

Table 8.3: Fastest key recovery with respect to the number of signatures.

Number of signatures	Probability of success (%)	Parameters			Total time
		BKZ	Preprocessing	Δ	
3	0.2	35	S_{11}	$\approx 2^3$	39 hours
4	4	35	S_{all}	$\approx 2^3$	25 hours 28
5	20	35	S_{all}	$\approx 2^3$	2 hours 42
6	40	35	S_{all}	$\approx 2^3$	1 hour 04
7	45	35	S_{all}	$\approx 2^3$	2 hours 36
8	45	35	S_{all}	$\approx 2^3$	5 hours 02

Table 8.4: Highest probability of success with respect to the number of signatures.

However, G6K allows to reduce lattices with much higher block-sizes than BKZ. For comparable probabilities of success, G6K is faster. Considering the highest probability achieved, on one hand, BKZ-35 leads to a probability of success of 45%, and a single reduction takes 133 minutes. On the other hand, to reach around the same probability of success with G6K, we increase the block-size to 80, and a single reduction is only around 45 minutes on average. This is an improvement by a factor of 3 in the reduction time.

Only 3 signatures. Using $\Delta \approx 2^3$ and no preprocessing, we recovered the secret key using 3 signatures with BKZ-35 only once and three times with BKZ-40. When using pre-processing S_{11}, BKZ-35 and $\Delta \approx 2^3$, the probability of success went up to 0.2%. Since all the probabilities remain much less than 1% an extensive analysis would have taken too long. Thus, in the rest of the section, the number of signatures only varies between 4 and 8. However, we want to emphasize that it is precisely this detailed analysis on a slightly higher number of signatures that allowed us to understand the impact of the parameters on the performance of the attack and resulted in finding the right ones allowing to mount the attack with 3 signatures.

Varying the bitsize of Δ. In Figure 8.1, we analyze the total time of key recovery as a function of the bitsize of Δ. We fix the block-size of BKZ to 25 and take traces without any preprocessing. We are able to recover the secret key by setting $\Delta = 1$, which is the lowest theoretical value one can choose. However, we observed a slight increase in the probability of success by taking a larger Δ. Without any surprise, we note that the total time to recover the secret key increases with the bitsize of Δ as the coefficients in the lattice basis become larger.

Analyzing the effect of preprocessing. We analyze the influence of our preprocessing method on the attack time. We fix BKZ block-size to 25. The effect of preprocessing is influenced by the bitsize of Δ and we give here an analysis for $\Delta \approx 2^{25}$ since the effect is more noticeable.

The effect of preprocessing is difficult to predict since its behavior varies depending on the parameters, having both positive and negative effects. On the one hand, we reduce the size of all the coefficients in the lattice, thus reducing the reduction time. On the other hand, we generate more potential small vectors[6] with norms smaller than the norm of $\mathbf{w}$. For this reason, the probability of success of the attack decreases, the vector $\mathbf{w}$ more likely to be a linear combination of vectors already in the reduced basis.

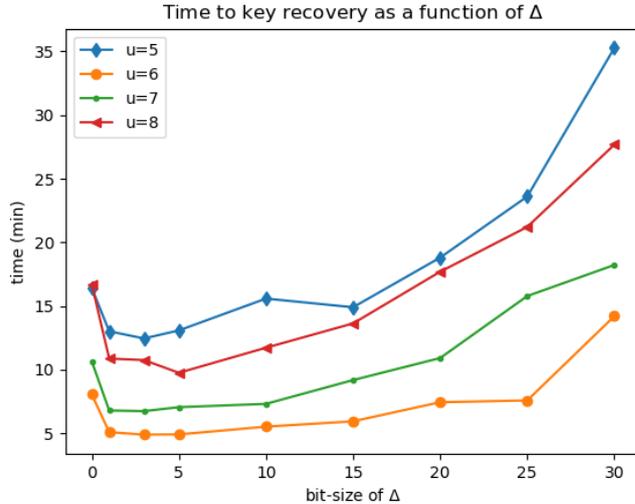[6]In the sense of vectors exhibited in (8.6).

Figure 8.1: Analyzing the overall time to recover the secret key as a function of the bitsize of Δ. We report the numbers BKZ-25 and no preprocessing. The optimal value for Δ is around 2^3 except for $u = 8$ where it is 2^5.

For example, with 7 signatures we find on average $\mathbf{w}$ to be the third or fourth vector in the reduced basis without preprocessing, whereas with S_{11} it is more likely to appear in position 40.

The positive effect of preprocessing is most noticeable for $u = 4$ and $u = 5$, as shown in Figure 8.2. For instance, using S_{15} and $u = 4$ lowers the overall time by a factor up to 5.7. For $u = 5$, we gain a factor close to 3 by using either S_{15} or S_{19}. For $u > 5$, using preprocessed traces is less impactful. For large Δ such as $\Delta \approx 2^{25}$, we still note some lower overall times when using S_{15} and S_{19}, up to a factor 2. When the bitsize gets smaller, reducing the size of the coefficients in the lattice is less impacful.

Balancing the block-size of BKZ. Finally, we vary the block-size in the BKZ algorithm. We fix $\Delta \approx 2^3$ and use no preprocessing. We plot the results in Figure 8.3 for 6 and 7 signatures. For other values of u, the plot is very similar and we omit them in Figure 8.3. Without any surprise, we see that as we increase the block-size, the probability of success increases, however the reduction time increases significantly as well. This explains the results shown in Table 8.3 and Table 8.4: to reach the best probability of success one needs to increase the block-size in BKZ (we did not try any block-size greater than 40), but to get the fastest key recovery attack, the block-size is chosen between 20 and 25, except for 3 signatures where the probability of success is too low with these parameters.

8.5 Error resilience analysis

It is not unexpected to have errors in the traces collected during side-channel attacks. Obtaining error-free traces requires some amount of work on the signal processing side. Prior to [DDME$^+$18], the presence of errors in traces was either ignored or preprocessing was done on the traces until an error-free sample was found, see [GPP$^+$16, ARAM17]. In [DDME$^+$18], it is shown the lattice attack still successfully recovers the secret key even when traces contain errors. An error in the setup given in [DDME$^+$18] corresponds to an incorrect bound on the size of the values being collected. In our setup, a trace without errors corresponds to a trace where every single coefficient in the wNAF representation of k has been identified correctly as either non-zero or not. The probability of having an error in our setup is thus much higher. Side-channel attacks without any errors are very rare. Both [vSY15] and [DDME$^+$18] give some analysis of the attacks Flush+Reload and Prime+Probe in real life scenarios.

In [FWC16], the results presented in the paper assume the Flush+Reload is implemented perfectly, without any error. In particular, to obtain 4 perfect traces and be able to run their experiment and find
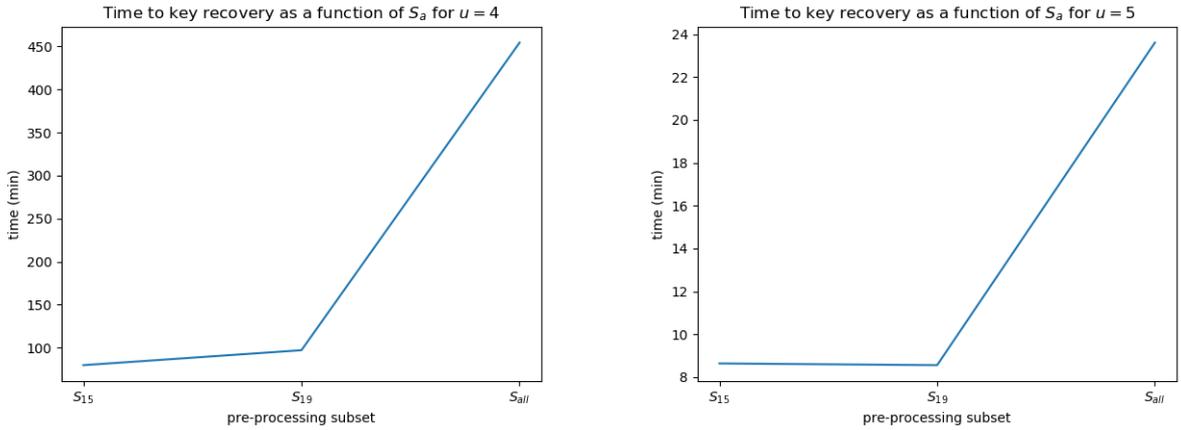
Figure 8.2: Overall time to recover the secret key as a function of the preprocessing subset for 4 and 5 traces. The other parameters are fixed: $\Delta \approx 2^{25}$ and BKZ-25.

Number of	Probability of success (%)				
signatures	0 error	5 errors	10 errors	20 errors	30 errors
4	0.28	$\ll 1$	0	0	0
5	4.58	0.86	0.18	$\ll 1$	0
6	19.52	5.26	1.26	0.14	$\ll 1$
7	33.54	10.82	3.42	0.32	$\ll 1$
8	35.14	13.26	4.70	0.58	$\ll 1$

Table 8.5: Error analysis using BKZ-25, $\Delta \approx 2^3$ and S_{all}.

the key, one would need to have in average 8 traces from Flush+Reload – the probability to conduct to a perfect reading of the traces being 56 % as pointed out in [vSY15]. In our work, we show that it is possible to recover the secret key using only 4, even erroneous, traces. However, the probability of success is very low.

Recall that an error in our case corresponds to a flipped digit in the trace of k. Table 8.5 shows the attack success probability in the presence of errors. We ran BKZ-25 and $\Delta \approx 2^3$ with traces taken from S_{all}. We average over 5000 experiments. We write $\ll 1$ when the attack succeeded less than five times over 5000 experiments, thus making it difficult to evaluate the probability of success.

The attack works up to a resilience to 2% of errors. Indeed, for $u = 6$, we recovered the secret key with 30 errors, *i.e.,* 30 flipped digits over 6×257 digits.

Different types of errors. There exists two possible types of errors. In the first case, a coefficient which is zero is evaluated as a non-zero coefficient. In theory, this only adds a new variable to the system, *i.e.,* the number ℓ of non-zero coefficients is overestimated. This does not affect the probability of success much. Indeed, we just have an overly-constrained system. We can see in Figure 8.4 that the probability of success of the attack indeed decreases slowly as we add errors of this form. With errors only of this form, we were able to recover the secret key up to nearly 4% of errors, (for instance with $u = 6$, using BKZ-35). The other type of errors consists of a non-zero coefficient which is misread as a zero coefficient. In this case, we lose information necessary for the key recovery and thus this type of error affects the probability of success far more importantly as can also be seen in Figure 8.4. In this setup, we were not able to recover the secret key when more than 3 errors of this type appear in the set of traces considered.
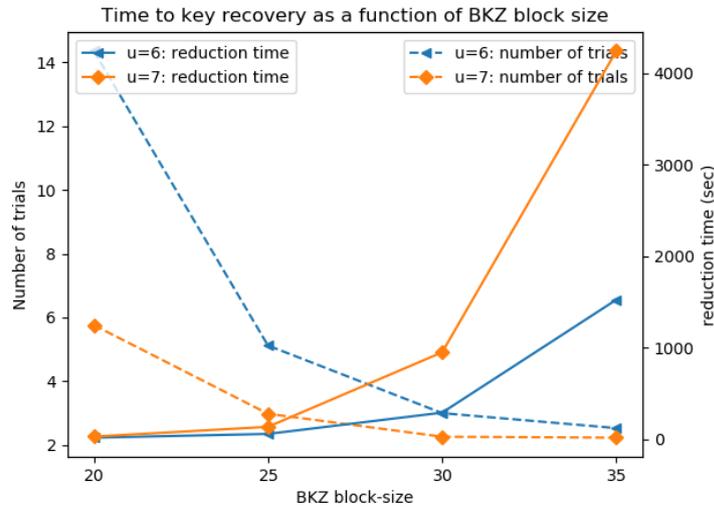
Figure 8.3: Analyzing the number of trials to recover the secret key and the reduction time of the lattice as a function of the block-size of BKZ. We consider the cases where $u = 6$ and $u = 7$. The dotted lines correspond to the number of trials, and the continued lines to the reduction time in seconds.
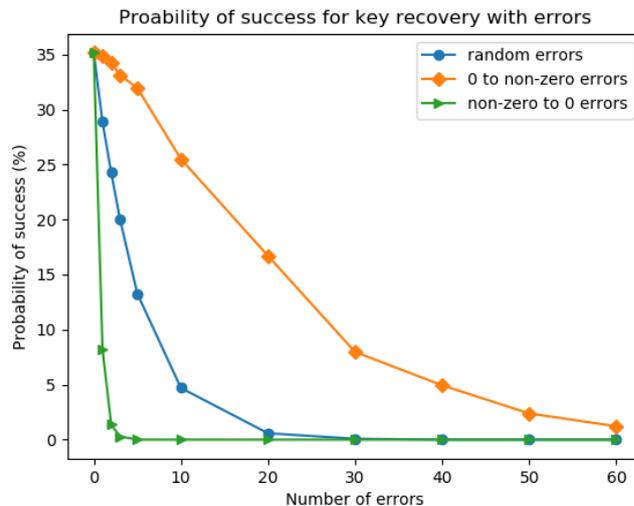

Figure 8.4: Probability of success for key recovery with various types of errors when using $u = 8$, BKZ-25, $\Delta \approx 2^3$, and no preprocessing.

Strategy. If the signal processing method is hesitant between a non-zero digit or 0, we would recommend to favor putting a non-zero instead of 0 to increase the chance of having an error of type $0 \rightarrow$ non-zero, for which the attack is a lot more tolerant.

8.5.1 Tables for error analysis

We analyze the effect of two possible kind of errors on the probability of success of our attack, using BKZ-25, $\Delta \approx 2^3$ and no preprocessing. We average over 5000 experiments. We write $\ll 1$ when the attack succeeded less than five times over 5000 experiments.

Number of	Probability of success (%)								
signatures	0 errors	1 error	5 errors	10 errors	20 errors	30 errors	40 errors	50 errors	60 errors
4	0.28	0.18	0.10	$\ll 1$	0	0	0	0	0
5	4.58	3.82	2.70	1.06	0.32	$\ll 1$	0	0	0
6	19.52	10.79	13.88	7.90	2.94	0.86	0.36	0.10	$\ll 1$
7	33.54	31.06	26.04	18.36	9.24	4.54	?	1.02	0.50
8	35.14	34.92	31.94	25.50	16.70	7.96	4.94	2.48	1.22

Table 8.6: Error $0 \to 1$ analysis using BKZ-25, $\Delta \approx 2^3$ and S_{all}.

Number of	Probability of success (%)			
signatures	0 errors	1 error	2 errors	3 errors
4	0.28	0	0	0
5	4.58	0.36	$\ll 1$	0
6	19.52	2.70	0.36	$\ll 1$
7	33.54	5.54	1.00	0.12
8	35.14	8.20	1.36	0.30

Table 8.7: Error $1 \to 0$ analysis using BKZ-25, $\Delta \approx 2^3$ and S_{all}.

When considering many errors, the probability of success can be increased by augmenting the block-size in the BKZ algorithm, as can be seen in Table 8.8.

Number of	Probability of success (%)															
signatures	30 errors				40 errors				50 errors				60 errors			
	25	30	35	40	25	30	35	40	25	30	35	40	25	30	35	40
5	$\ll 1$	0.24	0.35	0.75	0	$\ll 1$	$\ll 1$	0.42	0	0	0	0	0	0	0	0
6	0.86	2.48	3.58	3.97	0.36	0.90	1.18	2.28	0.10	0.36	0.58	0.94	$\ll 1$	$\ll 1$	0.12	0.12
7	4.54	6.44	7.32	8.73	1.80	3.54	3.48	4.58	1.02	1.26	1.84	3.26	0.50	0.62	1.20	1.43
8	7.96	10.46	11.78	10.98	4.94	6.12	6.73	7.12	2.48	3.26	3.78	4.64	1.22	1.84	1.89	2.18

Table 8.8: Errors $0 \to 1$ analysis with $\Delta \approx 2^3$, S_{all} and increasing block-size.

8.6   Countermeasures

In the last decades, most implementations of ECDSA have been the target of microarchitectural attacks, and thus existing implementations have either been replaced by more robust algorithms, or layers of security have been added. The work presented in this chapter does not fundamentally change the nature of these attacks but their performance. Thus known countermeasures for side-channel attacks still apply. For example, one way of minimizing leakage from the scalar multiplication is to use the Montgomery ladder scalar-by-point multiplication [Mon87], much more resilient to side-channel attacks due to the regularity of the operations. However, this does not entirely remove the risk of leakage [YB14]. Additional countermeasures are necessary.

When looking at common countermeasures, many implementations use blinding or masking techniques [OST06], for example in BouncyCastle implementation of ECDSA. The former consists in blinding the data before doing any operations, and masking techniques randomize all the data-dependent operations by applying random transformations, thus making any leakage unexploitable.

This work as well as any improvement on the lattice techniques that use traces from the side channel show that any leakage can be exploited. Hence, to prevent known attacks or even future ones, having implementations with zero leakage such as constant-time implementations should be the main goal.

Conclusion

The contributions presented in this thesis can be subdivided into two main topics: the evaluation of the hardness of the discrete logarithm problem over finite fields of medium and large characteristic and the analysis and use of methods to recover cryptographic keys from side-channel leakage due to implementation vulnerabilities. We summarize here the main contributions and give some insight on open questions and future directions.

On the hardness of the discrete logarithm problem

In this thesis, we focused on the discrete logarithm problem and in particular on the efficient algorithms known to solve it for specific finite fields.

We first considered the finite fields located at the boundary between small and medium characteristic. This area was particularly interesting to study as the complexities of the numerous algorithms that are concerned were non-existent in the literature. More importantly, it allowed us to give some insight on the security of pairing-based protocols. We were able to identify some special characteristics that are asymptotically as secure as characteristics of the same size but without any special form. This is a compelling fact as special characteristics were originally used to produce pairing-friendly elliptic curves. However, the threats of DLP attacks on the finite field side with SNFS have brought forth concerns about these special characteristics and the latter were then often avoided as they were thought to weaken the protocol. These special characteristics improve on the efficiency of the scheme and we show that in some cases they can be used safely.

Moreover, our asymptotic study aimed at complementing the practical security estimates given in [Gui20] for fixed security levels up to 192. The fact that some statements diverge, such as the aforementioned case of special characteristics, points to the fact that cryptanalists have not yet reached a steady state when considering a 192-bit level of security. More work for cryptographic relevant sizes, such as 256-bit of security level, is thus required in order to be confident on the security of the parameters proposed.

A first step towards having better security estimates would be to run large scale experiments with variants of the Number Field Sieve. This direction was partly achieved in this thesis where we present a first implementation of TNFS and a 521-bit record computation. However, more work on optimizing the many parameters involved would potentially lead to records using TNFS in much larger finite fields. We have also mentioned in Chapter 5 some potential speed-ups in both the relation collection step and the linear algebra step due to interactions between Galois actions and Schirokauer maps. Optimizing the parameters and taking into account Galois actions is left for future work. Establishing a new record computation for a large finite field is also left for future work.

Another research direction would be to explore the possibility of implementing another variant of NFS, the Multiple Number Field Sieve algorithm. Indeed, in theory, the use of multiple number fields significantly decreases the asymptotic complexity of NFS for finite fields we are concerned about. In practice, it is not clear whether this would allow to improve on actual computations. A potential implementation of MNFS would first require answering the following questions: how many number fields should be considered? What polynomial selection should be used? Moreover, it is possible that some steps of the algorithm must be adapted for this variant similarly as how sieving was modified for TNFS. In particular, the descent phase is likely to change for MNFS. Finally, having a practical MNFS algorithm would also require a large amount of implementation work similar to what has been done for TNFS.

In another direction, this thesis has not covered the Quasi-Polynomial algorithms used for finite fields of small characteristic. It is natural to wonder whether techniques from these algorithms can be translated to the medium and large case. A direct application of the main ideas of QP algorithms cannot be translated to medium and large characteristics since the complexity of these QP algorithms is in $L_{p^n}(a)$ when $p = L_{p^n}(a)$. Hence, as soon as $a > 1/3$, their complexity will exceed what we already have with NFS and its variants. However, one can still wonder whether some ideas can still be translated such as the use of function fields instead of number fields. Can we hope to go below an $L_{p^n}(1/3)$ complexity for finite fields of medium and large characteristic?

On key recovery methods from partial information

The second topic of our thesis concerned partial key recovery techniques using information from side-channel leakage. We first presented an overview of the most common techniques to recover a secret when partial information is first obtained from a side-channel attack. These techniques vary depending on the protocol considered and the nature of the information collected. Some open questions still persist concerning these techniques and further improvements to these methods should be possible.

First, recovering an RSA secret key when many non-consecutive chunks of bits of a factor of the RSA modulus need to recovered is still an open problem. When many non-consecutive bits of secret values are known or need to be recovered, the main method used is a branch and prune algorithm. However, when too many chunks are considered, the resulting tree has exponentially many solutions and thus the method is inefficient. In order to improve on this method and hence be able to recover RSA secrets with many unknown chunks of bits, additional information should be considered in order to efficiently prune the tree.

Another interesting and commonly used method is the Hidden Number Problem. The latter can be solved using two different techniques: via lattices or via Fourier analysis. The first method is usually favored as it requires less signatures, which is convenient for practical attacks. On the other hand, Fourier's analysis method can deal with as little as one known most significant bit from signature nonces. Combining these two methods to obtain "the best of both worlds" is an interesting research direction. A recent progress on the Hidden Number Problem can be found in [AH20].

Finally, we mention that recovering Diffie-Hellman secret key with multiple chunks of unknown bits is also still an open problem. Exploring multi-dimensional variants of the discrete logarithm problem seems like the right direction but so far has not proven to be practical due to boundary issues for the multi-dimensional pseudorandom walks.

In both attacks presented in Chapter 7 and Chapter 8, lattice techniques are used to recover the secret. In particular, in Chapter 8, we improve on the lattice construction used in the Extended Hidden Number Problem to minimize the number of signatures required for the algorithm to work, and also minimize the overall attack time. Further improving these attacks can thus also come from investigating more thoroughly the lattice constructions involved and the inputs to the lattice.

For example, we used a pre-processing technique in Chapter 8 which allowed us to save significant time in our attack. Similar pre-processing methods could be used for other techniques. Moreover, correctly balancing the coefficients in the lattices to efficiently find the desired short vector could also be improved and would consequently improve the success probability of these lattice attacks.

Finally, most of the attacks we concerned ourselves with resulted from a leakage coming from a fast modular exponentiation computation. Other operations in the execution of an algorithm could leak information. For example, in [CAPGATB19], the binary GCD step used during RSA key generation leaks information leading to a full RSA private key recovery. Even if real-world libraries now require their cryptographic software to be constant-time, it is unreasonable to believe that all protocols are safe from operations leaking information. Identifying these vulnerable operations and pieces of code that result in leaked bits and mounting full attacks to recover secret keys should remain an active field of research motivated by the wide use of these protocols in the real world.

Bibliography

[AAB+19] Frank Arute, Kunal Arya, Ryan Babbush, et al. Quantum supremacy using a programmable superconducting processor. *Nature*, 574:505–510, 2019.

[ABF+16] Thomas Allan, Billy Bob Brumley, Katrina Falkner, Joop van de Pol, and Yuval Yarom. Amplifying Side Channels through Performance Degradation. In *Proceedings of the 32nd Annual Conference on Computer Security Applications*, ACSAC'16, pages 422–435, New York, NY, USA, 2016. Association for Computing Machinery.

[ABF+20] Martin R. Albrecht, Shi Bai, Pierre-Alain Fouque, Paul Kirchner, Damien Stehlé, and Weiqiang Wen. Faster enumeration-based lattice reduction: Root hermite factor $k^{1/(2k)}$ time $k^{k/8+o(k)}$. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 186–212. Springer, Heidelberg, August 2020.

[ACMCC+18] Gora Adj, Isaac Canales-Martínez, Nareli Cruz-Cortés, Alfred Menezes, Thomaz Oliveira, Luis Rivera-Zamarripa, and Francisco Rodríguez-Henríquez. Computing Discrete Logarithms in Cryptographically-interesting Characteristic-three Finite Fields. *Advances in Mathematics of Communications*, 12:741–759, 2018.

[AD97] Miklós Ajtai and Cynthia Dwork. A Public-Key Cryptosystem with Worst-Case/Average-Case Equivalence. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC'97, pages 284–293, New York, NY, USA, 1997. Association for Computing Machinery.

[ADH+19] Martin R. Albrecht, Léo Ducas, Gottfried Herold, Elena Kirshanova, Eamonn W. Postlethwaite, and Marc Stevens. The general sieve kernel and new records in lattice reduction. Cryptology ePrint Archive, Report 2019/089, 2019. .

[Adl79] Leonard M. Adleman. A Subexponential Algorithm for the Discrete Logarithm Problem with Applications to Cryptography. In *20th Annual Symposium on Foundations of Computer Science (FOCS'79)*, pages 55–60. IEEE Computer Society Press, 1979.

[Adl94] Leonard M. Adleman. The Function Field Sieve. In Leonard M. Adleman and Ming-Deh Huang, editors, *ANTS-I*, LNCS, pages 108–121. Springer, 1994.

[AH20] Martin R. Albrecht and Nadia Heninger. On Bounded Distance Decoding with Predicate: Breaking the "Lattice Barrier" for the Hidden Number Problem. Cryptology ePrint Archive, Report 2020/1540, 2020. .

[Ajt96] Miklós Ajtai. Generating Hard Instances of Lattice Problems (Extended Abstract). In *Proceedings of the Twenty-Eighth Annual ACM Symposium on Theory of Computing*, STOC'96, pages 99–108, New York, NY, USA, 1996. Association for Computing Machinery.

[Ajt98] Miklós Ajtai. The Shortest Vector Problem in L2 is NP-Hard for Randomized Reductions (Extended Abstract). In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC'98, pages 10–19, New York, NY, USA, 1998. Association for Computing Machinery.

[Ajt03] Miklós Ajtai. The Worst-case Behavior of Schnorr's algorithm Approximating the Shortest Nonzero Vector in a Lattice. *Conference Proceedings of the Annual ACM Symposium on Theory of Computing*, pages 396–406, 2003.

[AKM+15] Marc Andrysco, David Kohlbrenner, Keaton Mowery, Ranjit Jhala, Sorin Lerner, and Hovav Shacham. On Subnormal Floating Point and Abnormal Timing. In *IEEE Symposium on Security and Privacy*, pages 623–639, 2015.

[AKS01] Miklós Ajtai, Ravi Kumar, and Dandapani Sivakumar. A Sieve Algorithm for the Shortest Lattice Vector Problem. New York, NY, USA, 2001. Association for Computing Machinery.

[AKS02] Miklós Ajtai, Ravi Kumar, and Dandapani Sivakumar. Sampling Short Lattice Vectors and the Closest Lattice Vector Problem. In *Proceedings 17th IEEE Annual Conference on Computational Complexity*, pages 53–57, 2002.

[AMG+15] Ittai Anati, Frank McKeen, Shay Gueron, Haitao Huang, Simon Johnson, Rebekah Leslie-Hurd, Harish Patil, Carlos Rozas, and Hisham Shafi. Intel Software Guard Extensions (Intel SGX). Tutorial Slides presented at ISCA, June 2015.

[ANT+20] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. LadderLeak: Breaking ECDSA with less than one bit of nonce leakage. In *ACM CCS 20*, pages 225–242. ACM Press, 2020.

[ARAM17] Jiji Angel, R. Rahul, C. Ashokkumar, and Bernard Menezes. DSA signing key recovery with noisy side channels and variable error rates. In Arpita Patra and Nigel P. Smart, editors, *INDOCRYPT 2017*, volume 10698 of *LNCS*, pages 147–165. Springer, Heidelberg, December 2017.

[AS08] Onur Aciiçmez and Werner Schindler. A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In Tal Malkin, editor, *CT-RSA 2008*, volume 4964 of *LNCS*, pages 256–273. Springer, Heidelberg, April 2008.

[ASK07] Onur Aciiçmez, Werner Schindler, and Çetin Kaya Koç. Cache based remote timing attack on the AES. In Masayuki Abe, editor, *CT-RSA 2007*, volume 4377 of *LNCS*, pages 271–286. Springer, Heidelberg, February 2007.

[ASM06] Man Ho Au, Willy Susilo, and Yi Mu. Constant-Size Dynamic k-TAA. In *SCN*, pages 111–125, 2006.

[Bab85] László Babai. On Lovász' Lattice Reduction and the Nearest Lattice Point Problem (Shortened Version). In *STACS*, 1985.

[Bar13] Razvan Barbulescu. *Algorithmes de logarithmes discrets dans les corps finis*. PhD thesis, 2013.

[BBG+17] Daniel J. Bernstein, Joachim Breitner, Daniel Genkin, Leon Groot Bruinderink, Nadia Heninger, Tanja Lange, Christine van Vredendaal, and Yuval Yarom. Sliding right into disaster: Left-to-right sliding windows leak. In Wieland Fischer and Naofumi Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 555–576. Springer, Heidelberg, September 2017.

[BBKZ16] Shi Bai, Cyril Bouvier, Alexander Kruppa, and Paul Zimmermann. Better polynomials for GNFS. *Mathematics of Computation*, 85(298):861–873, 2016.

[BBS04] Dan Boneh, Xavier Boyen, and Hovav Shacham. Short group signatures. In Matthew Franklin, editor, *CRYPTO 2004*, volume 3152 of *LNCS*, pages 41–55. Springer, Heidelberg, August 2004.

[BCC+13] Daniel J. Bernstein, Yun-An Chang, Chen-Mou Cheng, Li-Ping Chou, Nadia Heninger, Tanja Lange, and Nicko van Someren. Factoring RSA keys from certified smart cards: Coppersmith in the wild. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 341–360. Springer, Heidelberg, December 2013.

[BCD+17] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, Urs Müller, and Ahmad-Reza Sadeghi. DR. SGX: Hardening SGX Enclaves against Cache Attacks with Data Location Randomization. *arXiv preprint arXiv:1709.09917*, 2017.

[BCD+19] Ferdinand Brasser, Srdjan Capkun, Alexandra Dmitrienko, Tommaso Frassetto, Kari Kostiainen, and Ahmad-Reza Sadeghi. DR.SGX: Automated and Adjustable Side-Channel Protection for SGX Using Data Location Randomization. In *Proceedings of the 35th Annual Computer Security Applications Conference*, ACSAC'19, pages 788–800, New York, NY, USA, 2019. Association for Computing Machinery.

[BDF98] Dan Boneh, Glenn Durfee, and Yair Frankel. An attack on RSA given a small fraction of the private key bits. In Kazuo Ohta and Dingyi Pei, editors, *ASIACRYPT'98*, volume 1514 of *LNCS*, pages 25–34. Springer, Heidelberg, October 1998.

[BDGL16] Anja Becker, Léo Ducas, Nicolas Gama, and Thijs Laarhoven. New Directions in Nearest Neighbor Searching with Applications to Lattice Sieving. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'16, pages 10–24, USA, 2016. Society for Industrial and Applied Mathematics.

[Ber05] Daniel J. Bernstein. Cache-timing attacks on AES. Found at https://cr.yp.to/antiforgery/cachetiming-20050414.pdf, 2005.

[Ber08] Daniel J. Bernstein. Reducing lattice bases to find small-height values of univariate polynomials. *Surveys in Algorithmic Number Theory*, 44:421–446, 2008.

[Ber15] Elwyn R. Berlekamp. *Algebraic Coding Theory - Revised Edition*. World Scientific Publishing Co., Inc., USA, 2015.

[BF01] Dan Boneh and Matthew K. Franklin. Identity-based encryption from the Weil pairing. In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 213–229. Springer, Heidelberg, August 2001.

[BGG+20] Fabrice Boudot, Pierrick Gaudry, Aurore Guillevic, Nadia Heninger, Emmanuel Thomé, and Paul Zimmermann. Comparing the difficulty of factorization and discrete logarithm: A 240-digit experiment. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 62–91. Springer, Heidelberg, August 2020.

[BGGM15] Razvan Barbulescu, Pierrick Gaudry, Aurore Guillevic, and François Morain. Improving NFS for the discrete logarithm problem in non-prime finite fields. In Elisabeth Oswald and Marc Fischlin, editors, *EUROCRYPT 2015, Part I*, volume 9056 of *LNCS*, pages 129–155. Springer, Heidelberg, April 2015.

[BGJ15] Anja Becker, Nicolas Gama, and Antoine Joux. Speeding-up lattice sieving without increasing the memory, using sub-quadratic nearest neighbor search. Cryptology ePrint Archive, Report 2015/522, 2015. .

[BGJT14] Razvan Barbulescu, Pierrick Gaudry, Antoine Joux, and Emmanuel Thomé. A heuristic quasi-polynomial algorithm for discrete logarithm in finite fields of small characteristic. In Phong Q. Nguyen and Elisabeth Oswald, editors, *EUROCRYPT 2014*, volume 8441 of *LNCS*, pages 1–16. Springer, Heidelberg, May 2014.

[BGK15] Razvan Barbulescu, Pierrick Gaudry, and Thorsten Kleinjung. The tower number field sieve. In Tetsu Iwata and Jung Hee Cheon, editors, *ASIACRYPT 2015, Part II*, volume 9453 of *LNCS*, pages 31–55. Springer, Heidelberg, November / December 2015.

[BGS06] Ernie Brickell, Gary Graunke, and Jean-Pierre Seifert. Mitigating Cache/Timing Based Side-channels in AES and RSA Software Implementations. RSA Conference 2006 session DEV-203, February 2006.

[BL10] Yuval Bistritz and Alexander Lifshitz. Bounds for resultants of univariate and bivariate polynomials. *Linear Algebra and its Applications*, 432:1995–2005, 2010.

[BL11] Ernie Brickell and Jiangtao Li. Enhanced Privacy ID from Bilinear Pairing for Hardware Authentication and Attestation. *IJIPSI*, 1(1):3–33, 2011.

[BL13] Daniel J. Bernstein and Tanja Lange. Two Grumpy Giants and a Baby. In E.W. Howe and K.S. Kedlaya, editors, *ANTS X (Proceedings of the Tenth Algorithmic Number Theory Symposium, San Diego, California, July 9-13, 2012)*, The Open Book Series, pages 87–111, United States, 2013. Mathematical Sciences Publishers.

[BL14] Daniel J. Bernstein and Tanja Lange. Batch NFS. Cryptology ePrint Archive, Report 2014/921, 2014.

[BL16] Anja Becker and Thijs Laarhoven. Efficient (ideal) lattice sieving using cross-polytope LSH. In David Pointcheval, Abderrahmane Nitaj, and Tajjeeddine Rachidi, editors, *AFRICACRYPT 16*, volume 9646 of *LNCS*, pages 3–23. Springer, Heidelberg, April 2016.

[Ble98] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 1–12. Springer, Heidelberg, August 1998.

[BLS01] Dan Boneh, Ben Lynn, and Hovav Shacham. Short signatures from the Weil pairing. In Colin Boyd, editor, *ASIACRYPT 2001*, volume 2248 of *LNCS*, pages 514–532. Springer, Heidelberg, December 2001.

[BM03] Johannes Blömer and Alexander May. New partial key exposure attacks on RSA. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 27–43. Springer, Heidelberg, August 2003.

[BMD+17] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *WOOT*, 2017.

[Bon98] Dan Boneh. The decision Diffie-Hellman problem. In *Third Algorithmic Number Theory Symposium (ANTS)*, volume 1423 of *LNCS*. Springer, Heidelberg, 1998. Invited paper.

[Boo51] Andrew D. Booth. A Signed Binary Mutiplication Technique. *Q. J. Mech. Appl. Math.*, 4(2):236–240, January 1951.

[Bou15] Cyril Bouvier. *Algorithmes pour la factorisation d'entiers et le calcul de logarithme discret*. Theses, Université de Lorraine, June 2015.

[BP14] Razvan Barbulescu and Cécile Pierrot. The Multiple Number Field Sieve for Medium and High Characteristic Finite Fields. *LMS Journal of Computation and Mathematics*, 17:230–246, 2014.

[BR95] Mihir Bellare and Phillip Rogaway. Optimal asymmetric encryption. In Alfredo De Santis, editor, *EUROCRYPT'94*, volume 950 of *LNCS*, pages 92–111. Springer, Heidelberg, May 1995.

[BS84] László Babai and Endre Szemeredi. On The Complexity Of Matrix Group Problems I. In *25th Annual Symposium on Foundations of Computer Science, 1984*, pages 229–240, 1984.

[BT11] Billy Bob Brumley and Nicola Tuveri. Remote Timing Attacks Are Still Practical. In *ESORICS*, pages 355–371, 2011.

[BV96] Dan Boneh and Ramarathnam Venkatesan. Hardness of computing the most significant bits of secret keys in Diffie-Hellman and related schemes. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 129–142. Springer, Heidelberg, August 1996.

[BvSY14] Naomi Benger, Joop van de Pol, Nigel P. Smart, and Yuval Yarom. "ooh aah... just a little bit": A small amount of side channel can go a long way. In Lejla Batina and Matthew Robshaw, editors, *CHES 2014*, volume 8731 of *LNCS*, pages 75–92. Springer, Heidelberg, September 2014.

[BW14] David Bernhard and Bogdan Warinschi. *Cryptographic Voting — A Gentle Introduction*, pages 167–211. Springer International Publishing, Cham, 2014.

[cad] The CADO-NFS Development Team. CADO-NFS, An Implementation of the Number Field Sieve Algorithm. Found at http://cado-nfs.gforge.inria.fr. Development Version.

[CAPGATB19] Alejandro Cabrera Aldaya, Cesar Pereida García, Luis Manuel Alvarez Tapia, and Billy Bob Brumley. Cache-Timing Attacks on RSA Key Generation. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(4):213–242, Aug. 2019.

[CC03] Jae Choon Cha and Jung Hee Cheon. An identity-based signature from gap Diffie-Hellman groups. In Yvo Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 18–30. Springer, Heidelberg, January 2003.

[CD16] Victor Costan and Srinivas Devadas. Intel SGX explained. Cryptology ePrint Archive, Report 2016/086, 2016. .

[CEP83] Earl Rodney Canfield, Paul Erdős, and Carl Pomerance. On a Problem of Oppenheim concerning "Factorisation Numerorum". *Journal of number theory*, 17:1–28, 1983.

[CKP+19] Shaanan Cohney, Andrew Kwong, Shachar Paz, Daniel Genkin, Nadia Heninger, Eyal Ronen, and Yuval Yarom. Pseudorandom black swans: Cache attacks on CTR_DRBG. Cryptology ePrint Archive, Report 2019/996, 2019. .

[Coh12] Henri Cohen. *Advanced Topics in Computational Number Theory*. Graduate Texts in Mathematics. Springer New York, 2012.

[Cop84] Don Coppersmith. Fast evaluation of logarithms in fields of characteristic two. *IEEE Trans. Information Theory*, 30:587–594, 1984.

[Cop93] Don Coppersmith. Modifications to the number field sieve. *Journal of Cryptology*, 6(3):169–180, March 1993.

[Cop94] Don Coppersmith. Solving homogeneous linear equations over $GF(2)$ via block Wiedemann algorithm. *Mathematics of Computation*, 62:333–350, 1994.

[Cop96a] Don Coppersmith. Finding a small root of a bivariate integer equation; factoring with high bits known. In Ueli M. Maurer, editor, *EUROCRYPT'96*, volume 1070 of *LNCS*, pages 178–189. Springer, Heidelberg, May 1996.

[Cop96b] Don Coppersmith. Finding a small root of a univariate modular equation. In Ueli M. Maurer, editor, *EUROCRYPT'96*, volume 1070 of *LNCS*, pages 155–165. Springer, Heidelberg, May 1996.

[CS97]   Jan Camenisch and Markus Stadler. Efficient group signature schemes for large groups (extended abstract). In Burton S. Kaliski Jr., editor, *CRYPTO'97*, volume 1294 of *LNCS*, pages 410–424. Springer, Heidelberg, August 1997.

[CZRZ17]   Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. Detecting Privileged Side-Channel Attacks in Shielded Execution with Déjà Vu. In *AsiaCCS*, pages 7–18. ACM, 2017.

[DDME+18] Fergus Dall, Gabrielle De Micheli, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi, and Yuval Yarom. CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2018(2):171–191, May 2018.

[DGP20]   Gabrielle De Micheli, Pierrick Gaudry, and Cécile Pierrot. Asymptotic complexities of discrete logarithm algorithms in pairing-relevant finite fields. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part II*, volume 12171 of *LNCS*, pages 32–61. Springer, Heidelberg, August 2020.

[DH76]   Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Trans. Information Theory*, 22(6):644–654, 1976.

[DHMP13]   Elke De Mulder, Michael Hutter, Mark E. Marson, and Peter Pearson. Using Bleichenbacher's solution to the hidden number problem to attack nonce leaks in 384-bit ECDSA. In Guido Bertoni and Jean-Sébastien Coron, editors, *CHES 2013*, volume 8086 of *LNCS*, pages 435–452. Springer, Heidelberg, August 2013.

[Dic30]   Karl Dickman. On the Frequency of Numbers containing Prime Factors of a Certain Relative Magnitude. *Arkiv för matematik, astronomi och fysik*, 1930.

[DKTW19]   Jintai Ding, Seungki Kim, Tsuyoshi Takagi, and Yuntao Wang. LLL and stochastic sandpile models. Cryptology ePrint Archive, Report 2019/1009, 2019. .

[DPP20]   Gabrielle De Micheli, Rémi Piau, and Cécile Pierrot. A tale of three signatures: Practical attack of ECDSA with wNAF. In Abderrahmane Nitaj and Amr M. Youssef, editors, *AFRICACRYPT 20*, volume 12174 of *LNCS*, pages 361–381. Springer, Heidelberg, July 2020.

[DSvW21]   Léo Ducas, Marc Stevens, and Wessel van Woerden. Advanced Lattice Sieving on GPUs, with Tensor Cores. Cryptology ePrint Archive, Report 2021/141, 2021.

[Duc18]   Léo Ducas. Shortest vector from lattice sieving: A few dimensions for free. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part I*, volume 10820 of *LNCS*, pages 125–145. Springer, Heidelberg, April / May 2018.

[EL85]   Wim Van Eck and Neher Laborato. Electromagnetic radiation from video display units: An eavesdropping risk? *Computers and Security*, 4:269–286, 1985.

[ElG85]   Taher ElGamal. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. *IEEE Trans. Information Theory*, 31(4):469–472, 1985.

[FGHT17]   Joshua Fried, Pierrick Gaudry, Nadia Heninger, and Emmanuel Thomé. A kilobit hidden SNFS discrete logarithm computation. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 202–231. Springer, Heidelberg, April / May 2017.

[FH08]   Julie Ferrigno and Martin Hlavac. When AES blinks: Introducing optical side channel. *Information Security, IET*, 2:94 – 98, 10 2008.

[FK05]   Jens Franke and Thorsten Kleinjung. Continued Fractions and Lattice Sieving. *Special-Purpose Hardware for Attacking Cryptographic Systems–SHARCS*, page 40, 2005.

[FP85]   Ulrich Fincke and Michael Pohst. Improved Methods for Calculating Vectors of Short Length in a Lattice. *Mathematics of Computation*, 1985.

[FST10]   David Freeman, Michael Scott, and Edlyn Teske. A Taxonomy of Pairing-Friendly Elliptic Curves. *J. Cryptol.*, 23(2):224–280, April 2010.

[FWC16]   Shuqin Fan, Wenbo Wang, and Qingfeng Cheng. Attacking OpenSSL implementation of ECDSA with a few signatures. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1505–1515. ACM Press, October 2016.

[Gar59]   Harvey L. Garner. The residue number system. *IRE Trans. Electron. Computers*, EC-8(2):140–147, Jun 1959.

[GB17]   Cesar Pereida García and Billy Bob Brumley. Constant-time callees with variable-time callers. In Engin Kirda and Thomas Ristenpart, editors, *USENIX Security 2017*, pages 83–98. USENIX Association, August 2017.

[GBK11]   David Gullasch, Endre Bangerter, and Stephan Krenn. Cache Games – Bringing Access-Based Cache Attacks on AES to Practice. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP11, pages 490–505, USA, 2011. IEEE Computer Society.

[GBY16]   Cesar Pereida García, Billy Bob Brumley, and Yuval Yarom. "Make sure DSA signing exponentiations really are constant-time". In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1639–1650. ACM Press, October 2016.

[GGM17]   Laurent Grémy, Aurore Guillevic, and François Morain. Breaking DLP in $GF(p^5)$ using 3-dimensional sieving. July 2017.

[GGMT17] Laurent Grémy, Aurore Guillevic, François Morain, and Emmanuel Thomé. Computing discrete logarithms in $\mathbb{F}_{p^6}$. In Carlisle Adams and Jan Camenisch, editors, *SAC 2017*, volume 10719 of *LNCS*, pages 85–105. Springer, Heidelberg, August 2017.

[GGO+09] Vinodh Gopal, James Guilford, Erdinc Ozturk, Wajdi Feghali, Gil Wolrich, and Martin Dixon. Fast and Constant-Time Implementation of Modular Exponentiation. In *Embedded Systems and Communications Security*, September 2009.

[GKL+20] Robert Granger, Thorsten Kleinjung, Arjen K. Lenstra, Benjamin Wesolowski, and Jens Zumbragel. Computation of a 30750-bit binary field discrete logarithm. Cryptology ePrint Archive, Report 2020/965, 2020. .

[GKZ14]   Robert Granger, Thorsten Kleinjung, and Jens Zumbrägel. Breaking '128-bit secure' supersingular binary curves - (or how to solve discrete logarithms in $\mathbb{F}_{2^{4\cdot1223}}$ and $\mathbb{F}_{2^{12\cdot367}}$). In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part II*, volume 8617 of *LNCS*, pages 126–145. Springer, Heidelberg, August 2014.

[GKZ18]   Robert Granger, Thorsten Kleinjung, and Jens Zumbrägel. Indiscreet logarithms in finite fields of small characteristic. *Advances in Mathematics of Communications*, 12:263–286, 2018.

[GM03]   Daniel Goldstein and Andrew Mayer. On the equidistribution of Hecke points. *Forum Mathematicum*, 15:165–189, 01 2003.

[GMM16]   Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *DIMVA*, pages 300–321, 2016.

[GMSS99] Oded Goldreich, Daniele Micciancio, Shmuel Safra, and Jean-Pierre Seifert. Approximating Shortest Lattice Vectors is Not Harder than Approximating Closet Lattice Vectors. *Inf. Process. Lett.*, 71(2):55–61, July 1999.

[GMT16] Aurore Guillevic, François Morain, and Emmanuel Thomé. Solving discrete logarithms on a 170-bit MNT curve by pairing reduction. In Roberto Avanzi and Howard M. Heys, editors, *SAC 2016*, volume 10532 of *LNCS*, pages 559–578. Springer, Heidelberg, August 2016.

[GMWM16] Daniel Gruss, Clémentine Maurice, Klaus Wagner, and Stefan Mangard. Flush+flush: A fast and stealthy cache attack. In *Proceedings of the 13th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment - Volume 9721*, DIMVA 2016, pages 279–299, Berlin, Heidelberg, 2016. Springer-Verlag.

[GN08] Nicolas Gama and Phong Q. Nguyen. Predicting lattice reduction. In Nigel P. Smart, editor, *EUROCRYPT 2008*, volume 4965 of *LNCS*, pages 31–51. Springer, Heidelberg, April 2008.

[GNR10] Nicolas Gama, Phong Q. Nguyen, and Oded Regev. Lattice enumeration using extreme pruning. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 257–278. Springer, Heidelberg, May / June 2010.

[Gor93] Daniel Gordon. Discrete Logarithms in $GF(P)$ Using the Number Field Sieve. *SIAM Journal on Discrete Mathematics*, 6:124–138, 1993.

[Gor98] Daniel M. Gordon. A survey of fast exponentiation methods. *J. Algorithms*, 27(1):129–146, April 1998.

[GPP+16] Daniel Genkin, Lev Pachmanov, Itamar Pipman, Eran Tromer, and Yuval Yarom. ECDSA key extraction from mobile devices via nonintrusive physical side channels. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi, editors, *ACM CCS 2016*, pages 1626–1638. ACM Press, October 2016.

[Gré16] Laurent Grémy. Computations of Discrete Logarithms. Found at https://dldb.loria.fr, 2016.

[Gré18] Laurent Grémy. Higher dimensional sieving for the number field sieve algorithms. In *ANTS 2018 - Thirteenth Algorithmic Number Theory Symposium*, pages 1–16, Madison, United States, July 2018. University of Wisconsin.

[GS21] Aurore Guillevic and Shashank Singh. On the alpha value of polynomials in the Tower Number Field Sieve Algorithm. *Mathematical Cryptology*, 1(1), Feb. 2021.

[GSM15] Daniel Gruss, Raphael Spreitzer, and Stefan Mangard. Cache template attacks: Automating attacks on inclusive last-level caches. In Jaeyeon Jung and Thorsten Holz, editors, *USENIX Security 2015*, pages 897–912. USENIX Association, August 2015.

[GST14] Daniel Genkin, Adi Shamir, and Eran Tromer. RSA key extraction via low-bandwidth acoustic cryptanalysis. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 444–461. Springer, Heidelberg, August 2014.

[Gui] Aurore Guillevic. Pairing-friendly curves. Blogpost found at https://members.loria.fr/AGuillevic/pairing-friendly-curves.

[Gui19] Aurore Guillevic. Faster individual discrete logarithms in finite fields of composite extension degree. *Mathematics of Computation*, 88(317):1273–1301, January 2019.

[Gui20] Aurore Guillevic. A short-list of pairing-friendly curves resistant to special TNFS at the 128-bit security level. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020, Part II*, volume 12111 of *LNCS*, pages 535–564. Springer, Heidelberg, May 2020.

[GWZ17] Steven D. Galbraith, Ping Wang, and Fangguo Zhang. Computing Elliptic Curve Discrete Logarithms with Improved Baby-step Giant-step Algorithm. *Advances in Mathematics of Communications*, 11:453, 2017.

[GZES17] Berk Gülmezoglu, Andreas Zankl, Thomas Eisenbarth, and Berk Sunar. PerfWeb: How to Violate Web Privacy with Hardware Performance Events. In *ESORICS (2)*, pages 80–97, 2017.

[HG97] Nicholas Howgrave-Graham. Finding small roots of univariate modular equations revisited. In Michael Darnell, editor, *Crytography and Coding*, pages 131–142, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[HG98] Nicholas A Howgrave-Graham. *Computational Mathematics Inspired by RSA*. PhD thesis, University of Bath, 1998.

[HG01] Nick Howgrave-Graham. Approximate integer common divisors. In *International Cryptography and Lattices Conference*, pages 51–66. Springer, 2001.

[HKL18] Gottfried Herold, Elena Kirshanova, and Thijs Laarhoven. Speed-ups and time-memory trade-offs for tuple lattice sieving. In Michel Abdalla and Ricardo Dahab, editors, *PKC 2018, Part I*, volume 10769 of *LNCS*, pages 407–436. Springer, Heidelberg, March 2018.

[HM08] Mathias Herrmann and Alexander May. Solving linear equations modulo divisors: On factoring given any bits. In Josef Pieprzyk, editor, *ASIACRYPT 2008*, volume 5350 of *LNCS*, pages 406–424. Springer, Heidelberg, December 2008.

[HPS98] Jeffrey Hoffstein, Jill Pipher, and Joseph H. Silverman. NTRU: A Ring-based Public Key Cryptosystem. In Joe P. Buhler, editor, *Algorithmic Number Theory*, pages 267–288, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[HPS11a] Guillaume Hanrot, Xavier Pujol, and Damien Stehlé. Algorithms for the Shortest and Closest Lattice Vector Problems. In Yeow Meng Chee, Zhenbo Guo, San Ling, Fengjing Shao, Yuansheng Tang, Huaxiong Wang, and Chaoping Xing, editors, *Coding and Cryptology*, pages 159–190, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[HPS11b] Guillaume Hanrot, Xavier Pujol, and Damien Stehlé. Analyzing blockwise lattice algorithms using dynamical systems. In Phillip Rogaway, editor, *CRYPTO 2011*, volume 6841 of *LNCS*, pages 447–464. Springer, Heidelberg, August 2011.

[HR07] Martin Hlavác and Tomás Rosa. Extended hidden number problem and its cryptanalytic applications. In Eli Biham and Amr M. Youssef, editors, *SAC 2006*, volume 4356 of *LNCS*, pages 114–133. Springer, Heidelberg, August 2007.

[HS07] Guillaume Hanrot and Damien Stehlé. Improved analysis of kannan's shortest lattice vector algorithm. In Alfred Menezes, editor, *CRYPTO 2007*, volume 4622 of *LNCS*, pages 170–186. Springer, Heidelberg, August 2007.

[HS09] Nadia Heninger and Hovav Shacham. Reconstructing RSA private keys from random key bits. In Shai Halevi, editor, *CRYPTO 2009*, volume 5677 of *LNCS*, pages 1–17. Springer, Heidelberg, August 2009.

[HS14] Michael Hutter and Jörn-Marc Schmidt. The temperature side channel and heating fault attacks. Cryptology ePrint Archive, Report 2014/190, 2014. .

[HSH+08] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In Paul C. van Oorschot, editor, *USENIX Security 2008*, pages 45–60. USENIX Association, July / August 2008.

[IAES15] Gorka Irazoqui Apecechea, Thomas Eisenbarth, and Berk Sunar. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing - and Its Application to AES. In *IEEE Symposium on Security and Privacy*, pages 591–604, 2015.

[IEE00] IEEE. Minutes from the IEEE P1363 Working Group for Public-Key Cryptography Standards, November 2000.

[IGI+15] Mehmet Sinan Inci, Berk Gulmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Seriously, get off my cloud! Cross-VM RSA key recovery in a public cloud. Cryptology ePrint Archive, Report 2015/898, 2015. .

[IGI+16] Mehmet Sinan Inci, Berk Gülmezoglu, Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. Cache Attacks Enable Bulk Key Recovery on the Cloud. In *CHES*, pages 368–388, 2016.

[Int09] International Organization for Standardization. Information Technology - Security Techniques – Cryptographic techniques based on elliptic curves. Part 5: Elliptic curve generation, 2009.

[JL03] Antoine Joux and Reynald Lercier. Improvements to the general number field sieve for discrete logarithms in prime fields. a comparison with the gaussian integer method. *Mathematics of Computation*, 72:953–967, 2003.

[JL06] Antoine Joux and Reynald Lercier. The function field sieve in the medium prime case. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 254–270. Springer, Heidelberg, May / June 2006.

[JLSV06] Antoine Joux, Reynald Lercier, Nigel Smart, and Frederik Vercauteren. The number field sieve in the medium prime case. In Cynthia Dwork, editor, *CRYPTO 2006*, volume 4117 of *LNCS*, pages 326–344. Springer, Heidelberg, August 2006.

[JMV01] Don Johnson, Alfred Menezes, and Scott Vanstone. The Elliptic Curve Digital Signature Algorithm (ECDSA). *International Journal of Information Security*, 1(1):36–63, 2001.

[Jou13] Antoine Joux. Faster index calculus for the medium prime case application to 1175-bit and 1425-bit finite fields. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 177–193. Springer, Heidelberg, May 2013.

[Jou14] Antoine Joux. A new index calculus algorithm with complexity $L(1/4 + o(1))$ in small characteristic. In Tanja Lange, Kristin Lauter, and Petr Lisonek, editors, *SAC 2013*, volume 8282 of *LNCS*, pages 355–379. Springer, Heidelberg, August 2014.

[JP14a] Antoine Joux and Cécile Pierrot. Improving the polynomial time precomputation of frobenius representation discrete logarithm algorithms - simplified setting for small characteristic finite fields. In Palash Sarkar and Tetsu Iwata, editors, *ASIACRYPT 2014, Part I*, volume 8873 of *LNCS*, pages 378–397. Springer, Heidelberg, December 2014.

[JP14b] Antoine Joux and Cécile Pierrot. The special number field sieve in $\mathbb{F}_{p^n}$ - application to pairing-friendly constructions. In Zhenfu Cao and Fangguo Zhang, editors, *PAIRING 2013*, volume 8365 of *LNCS*, pages 45–61. Springer, Heidelberg, November 2014.

[JP16]   Antoine Joux and Cécile Pierrot. Nearly Sparse Linear Algebra and application to Discrete Logarithms Computations. In *Contemporary Developments in Finite Fields and Applications* . 2016.

[JP19]   Antoine Joux and Cecile Pierrot. Algorithmic aspects of elliptic bases in finite field discrete logarithm algorithms. Cryptology ePrint Archive, Report 2019/782, 2019. .

[Kal95]   Erich Kaltofen. Analysis of Coppersmith's Block Wiedemann Algorithm for the Parallel Solution of Sparse Linear Systems. *Mathematics of Computation*, 64(210):777–806, 1995.

[Kal97]   Michael Kalkbrener. An upper bound on the number of monomials in determinants of sparse matrices with symbolic entries. *Mathematica Pannonica*, 8:73–82, 1997.

[Kan83]   Ravi Kannan. Improved Algorithms for Integer Programming and Related Lattice Problems. In *Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, STOC'83, pages 193–206, New York, NY, USA, 1983. Association for Computing Machinery.

[Kan87]   Ravi Kannan. Minkowski's Convex Body Theorem and Integer Programming. *Mathematics of Operations Research*, 12:415–440, 1987.

[KB16]   Taechan Kim and Razvan Barbulescu. Extended tower number field sieve: A new complexity for the medium prime case. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 543–571. Springer, Heidelberg, August 2016.

[KDL+17]   Thorsten Kleinjung, Claus Diem, Arjen K. Lenstra, Christine Priplata, and Colin Stahlke. Computation of a 768-bit prime field discrete logarithm. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *EUROCRYPT 2017, Part I*, volume 10210 of *LNCS*, pages 185–201. Springer, Heidelberg, April / May 2017.

[KEF19]   Paul Kirchner, Thomas Espitau, and Pierre-Alain Fouque. Algebraic and euclidean lattices: Optimal lattice reduction and beyond. Cryptology ePrint Archive, Report 2019/1436, 2019. .

[KGG+18]   Paul Kocher, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. *arXiv preprint arXiv:1801.01203*, 2018.

[KHF+19]   Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy*, pages 1–19. IEEE Computer Society Press, May 2019.

[KJ17]   Taechan Kim and Jinhyuck Jeong. Extended tower number field sieve with application to finite fields of arbitrary composite extension degree. In Serge Fehr, editor, *PKC 2017, Part I*, volume 10174 of *LNCS*, pages 388–408. Springer, Heidelberg, March 2017.

[KJJ99]   Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 388–397. Springer, Heidelberg, August 1999.

[KJJR11]   Paul Kocher, Joshua Jaffe, Benjamin Jun, and Pankaj Rohatgi. Introduction to differential power analysis. *Journal of Cryptographic Engineering*, 1(1):5–27, Apr 2011.

[KM05]   Neal Koblitz and Alfred Menezes. Pairing-based Cryptography at High Security Levels. In *Proceedings of Cryptography and Coding 2005, volume 3796 of LNCS*, pages 13–36. Springer-Verlag, 2005.

[KM07] Neal Koblitz and Alfred Menezes. Another Look at Generic Groups. *Advances in Mathematics of Communications*, 1:13, 2007.

[Knu97] Donald E. Knuth. *The Art of Computer Programming, Volume 2 (3rd Ed.): Seminumerical Algorithms*. Addison-Wesley Longman Publishing Co., Inc., USA, 1997.

[Koc96] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 104–113. Springer, Heidelberg, August 1996.

[Kra22] Maurice Kraitchik. *Théorie des nombres*. Number v. 1 in Théorie des nombres. Gauthier-Villars, 1922.

[KW19] Thorsten Kleinjung and Benjamin Wesolowski. Discrete logarithms in quasi-polynomial time in finite fields of fixed characteristic. Cryptology ePrint Archive, Report 2019/751, 2019. .

[Laa16] Thijs Laarhoven. Sieving for closest lattice vectors (with preprocessing). In Roberto Avanzi and Howard M. Heys, editors, *SAC 2016*, volume 10532 of *LNCS*, pages 523–542. Springer, Heidelberg, August 2016.

[Len87] Hendrik W. Lenstra. Factoring Integers with Elliptic Curves. *Annals of Mathematics*, 126(3):649–673, 1987.

[LGS$^+$16] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. ARMageddon: Cache attacks on mobile devices. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 549–564. USENIX Association, August 2016.

[LGS$^+$17] Moritz Lipp, Daniel Gruss, Michael Schwarz, David Bidner, Clémentine Maurice, and Stefan Mangard. Practical Keystroke Timing Attacks in Sandboxed JavaScript. In *ESORICS (2)*, pages 191–209, 2017.

[LLL82] Arjen Klaas Lenstra, Hendrik Willem Lenstra, and László Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.

[LM18] Thijs Laarhoven and Artur Mariano. Progressive Lattice Sieving. In *International Conference on Post-Quantum Cryptography*, pages 292–311. Springer, 2018.

[LN20] Jianwei Li and Phong Q. Nguyen. A complete analysis of the BKZ lattice reduction algorithm. Cryptology ePrint Archive, Report 2020/1237, 2020. .

[LPR10] Vadim Lyubashevsky, Chris Peikert, and Oded Regev. On ideal lattices and learning with errors over rings. In Henri Gilbert, editor, *EUROCRYPT 2010*, volume 6110 of *LNCS*, pages 1–23. Springer, Heidelberg, May / June 2010.

[LPSW19] Changmin Lee, Alice Pellet-Mary, Damien Stehlé, and Alexandre Wallet. An LLL algorithm for module lattices. In Steven D. Galbraith and Shiho Moriai, editors, *ASIACRYPT 2019, Part II*, volume 11922 of *LNCS*, pages 59–90. Springer, Heidelberg, December 2019.

[LS15] Adeline Langlois and Damien Stehlé. Worst-Case to Average-Case Reductions for Module Lattices. *Des. Codes Cryptography*, 75(3):565–599, June 2015.

[LSG$^+$17] Sangho Lee, Ming-Wei Shih, Prasun Gera, Taesoo Kim, Hyesoon Kim, and Marcus Peinado. Inferring Fine-grained Control Flow Inside SGX Enclaves with Branch Shadowing. In *USENIX Security Symposium*, pages 557–574, 2017.

[LSG+18] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 973–990. USENIX Association, August 2018.

[LYG+15] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. Last-Level Cache Side-Channel Attacks are Practical. In *IEEE Symposium on Security and Privacy*, pages 605–622, 2015.

[Mas69] James L. Massey. Shift-register Synthesis and BCH Decoding. *IEEE Transactions on Information Theory*, 15(1):122–127, 1969.

[Mat03] Dmitri V. Matyukhin. On asymptotic complexity of computing discrete logarithms over $GF(p)$. *Discrete Mathematics and Applications*, 13:27–50, 2003.

[May10] Alexander May. Using LLL-reduction for solving RSA and factorization problems. ISC, pages 315–348. Springer, Heidelberg, 2010.

[MBA+20] Robert Merget, Marcus Brinkmann, Nimrod Aviram, Juraj Somorovsky, Johannes Mittmann, and Jörg Schwenk. Raccoon Attack: Finding and Exploiting Most-Significant-Bit-Oracles in TLS-DH(E). 2020.

[Mer78] Ralph C. Merkle. Secure Communications over Insecure Channels. *Commun. ACM*, 21(4):294–299, April 1978.

[MES18] Ahmad Moghimi, Thomas Eisenbarth, and Berk Sunar. MemJam: A false dependency attack against constant-time crypto implementations in SGX. In Nigel P. Smart, editor, *CT-RSA 2018*, volume 10808 of *LNCS*, pages 21–44. Springer, Heidelberg, April 2018.

[MG02] Daniele Micciancio and Shafi. Goldwasser. *Complexity of Lattice Problems*. Kluwer Academic Publishers, USA, 2002.

[Mic04] Daniele Micciancio. Generalized Compact Knapsacks, Cyclic Lattices, and Efficient One-way Functions from Worst-case Complexity Assumptions. *Electronic Colloquium on Computational Complexity (ECCC)*, 01 2004.

[Mic11] Daniele Micciancio. How-many-lll-reduced-bases-are-there? answer1. Overflow: http://mathoverflow.net, 2011.

[MIE17] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. CacheZoom: How SGX amplifies the power of cache attacks. In Wieland Fischer and Naofumi Homma, editors, *CHES 2017*, volume 10529 of *LNCS*, pages 69–90. Springer, Heidelberg, September 2017.

[Mil75] Jeffrey C. P. Miller. On Factorisation, with a Suggested New Approach. *Mathematics of Computation*, 29(129):155–172, 1975.

[MLB17] Artur Mariano, Thijs Laarhoven, and Christian Bischof. A Parallel Variant of LDSieve for the SVP on Lattices. In *2017 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP)*, pages 23–30. IEEE, 2017.

[MN01] Atsuko Miyaji and Masaki Nakabayashi. New Explicit Conditions of Elliptic Curve Traces for FR-Reduction. *IEEE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E84-A(5):1234–1243, May 2001.

[Möl03] Bodo Möller. Improved techniques for fast exponentiation. In Pil Joong Lee and Chae Hoon Lim, editors, *ICISC 02*, volume 2587 of *LNCS*, pages 298–312. Springer, Heidelberg, November 2003.

[Mon87] Peter L. Montgomery. Speeding the Pollard and elliptic curve methods of factorization. *Mathematics of Computation*, 48(177):243–243, jan 1987.

[MR21] Gary McGuire and Oisín Robinson. Lattice Sieving in Three Dimensions for Discrete Log in Medium Characteristic. *Journal of Mathematical Cryptology*, 15(1):223 – 236, 01 Jan. 2021.

[MSST20] Madhurima Mukhopadhyay, Palash Sarkar, Shashank Singh, and Emmanuel Thomé. New Discrete Logarithm Computation for the Medium Prime Case using the Function Field Sieve. *Advances in Mathematics of Communications*, 0, 2020.

[MV10a] Manfred Madritsch and Brigitte Vallée. Modelling the LLL Algorithm by Sandpiles. In *Proceedings of the 9th Latin American Conference on Theoretical Informatics*, LATIN'10, pages 267–281, Berlin, Heidelberg, 2010. Springer-Verlag.

[MV10b] Daniele Micciancio and Panagiotis Voulgaris. A Deterministic Single Exponential Time Algorithm for Most Lattice Problems Based on Voronoi Cell Computations. In *Proceedings of the Forty-Second ACM Symposium on Theory of Computing*, STOC '10, pages 351–358, New York, NY, USA, 2010. Association for Computing Machinery.

[MV10c] Daniele Micciancio and Panagiotis Voulgaris. Faster Exponential Time Algorithms for the Shortest Vector Problem. In *Proceedings of the twenty-first annual ACM-SIAM symposium on Discrete Algorithms*, pages 1468–1480. SIAM, 2010.

[MVH+20] Daniel Moghimi, Jo Van Bulck, Nadia Heninger, Frank Piessens, and Berk Sunar. Copy-Cat: Controlled instruction-level attacks on enclaves. In Srdjan Capkun and Franziska Roesner, editors, *USENIX Security 2020*, pages 469–486. USENIX Association, August 2020.

[MW68] Jeffrey C.P. Miller and Alfred E. Western. *Tables of Indices and Primitive Roots*. University Press, 1968.

[MW96] Ueli M. Maurer and Stefan Wolf. Diffie-Hellman oracles. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 268–282. Springer, Heidelberg, August 1996.

[MW15] Daniele Micciancio and Michael Walter. Fast Lattice Point Enumeration with Minimal Overhead. In *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA'15, pages 276–294, USA, 2015. Society for Industrial and Applied Mathematics.

[Nec94] V. I. Nechaev. Complexity of a determinate algorithm for the discrete logarithm. *Mathematical Notes*, 55(2):165–172, 1994.

[NIS13] National Institute of Standards and Technology. *Digital Signature Standard (DSS)*, 2013.

[NS02] Phong Q. Nguyen and Igor E. Shparlinski. The Insecurity of the Digital Signature Algorithm with Partially Known Nonces. *J. Cryptology*, 15(3):151–176, 2002.

[NS03] Phong Q. Nguyen and Igor E. Shparlinski. The Insecurity of the Elliptic Curve Digital Signature Algorithm with Partially Known Nonces. *Des. Codes Cryptography*, 30(2):201–217, 2003.

[NS06] Phong Q. Nguyen and Damien Stehlé. LLL on the Average. In *Proceedings of the 7th International Conference on Algorithmic Number Theory*, ANTS06, Berlin, Heidelberg, 2006. Springer-Verlag.

[NV08] Phong Nguyen and Thomas Vidick. Sieve Algorithms for the Shortest Vector Problem are Practical. *Journal of Mathematical Cryptology*, 2:181–207, 07 2008.

[OKSK15] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *CCS*, pages 1406–1418. ACM, 2015.

[OST06] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *CT-RSA 2006*, volume 3860 of *LNCS*, pages 1–20. Springer, Heidelberg, February 2006.

[OW99] Paul C. Oorschot and Michael J. Wiener. Parallel collision search with cryptanalytic applications. *J. Cryptol.*, 12(1):1–28, January 1999.

[Pag02] Dan Page. Theoretical Use of Cache Memory as a Cryptanalytic Side-Channel. IACR Cryptology ePrint Archive, Report 2002/169, 2002.

[Per05] Colin Percival. Cache Missing for Fun and Profit. In *BSDCon 2005*, Ottawa, CA, 2005.

[PGF98] Daniel Panario, Xavier Gourdon, and Philippe Flajolet. An analytic approach to smooth polynomials over finite fields. In Joe P. Buhler, editor, *ANTS-III*, LNCS, pages 226–236. Springer, 1998.

[PH78] Stephen C. Pohlig and Martin Hellman. An improved algorithm for computing logarithms over $GF(p)$ and its cryptographic significance (corresp.). *IEEE Transactions on Information Theory*, 24(1):106–110, 1978.

[PHS19] Alice Pellet-Mary, Guillaume Hanrot, and Damien Stehlé. Approx-SVP in ideal lattices with pre-processing. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part II*, volume 11477 of *LNCS*, pages 685–716. Springer, Heidelberg, May 2019.

[Poh81] Michael Pohst. On the Computation of Lattice Vectors of Minimal Length, Successive Minima and Reduced Bases with Applications. *SIGSAM Bull.*, 15(1):37–44, February 1981.

[Pol78] John M. Pollard. Monte Carlo methods for index computation mod p. *Mathematics of Computation*, 32:918–924, 1978.

[Pol93] John M. Pollard. The Lattice Sieve. In Arjen K. Lenstra and Hendrik W. Lenstra, editors, *The development of the number field sieve*, pages 43–49, Berlin, Heidelberg, 1993. Springer Berlin Heidelberg.

[Pom87] Carl Pomerance. Fast, rigorous factorization and discrete logarithm algorithms. *Discrete algorithms and complexity*, pages 119–143, 1987.

[PPS12] Kenneth G. Paterson, Antigoni Polychroniadou, and Dale L. Sibborn. A coding-theoretic approach to recovering noisy RSA keys. In Xiaoyun Wang and Kazue Sako, editors, *ASIACRYPT 2012*, volume 7658 of *LNCS*, pages 386–403. Springer, Heidelberg, December 2012.

[PS09] Xavier Pujol and Damien Stehlé. Solving the shortest lattice vector problem in time $2^{2.465n}$. Cryptology ePrint Archive, Report 2009/605, 2009. .

[PS13] Thomas Plantard and Michael Schneider. Creating a challenge for ideal lattices. Cryptology ePrint Archive, Report 2013/039, 2013. .

[QS01] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *Proceedings of the International Conference on Research in Smart Cards: Smart Card Programming and Security*, E-SMART '01, pages 200–210, London, UK, 2001. Springer-Verlag.

[RSA78] Ron L. Rivest, Adi Shamir, and Leonard Adleman. A Method for Obtaining Digital Signatures and Public-Key Cryptosystems. *Commun. ACM*, 21(2):120–126, February 1978.

[Rup10] Raminder Singh Ruprai. *Improvements to the Gaudry-Schost Algorithm for Multidimensional discrete logarithm problems and Applications*. PhD thesis, 2010.

[Sch86] Claus-Peter Schnorr. A more efficient algorithm for lattice basis reduction. In Laurent Kott, editor, *Automata, Languages and Programming*, pages 359–369, Berlin, Heidelberg, 1986. Springer Berlin Heidelberg.

[Sch87] Claus-Peter Schnorr. A Hierarchy of Polynomial Time Basis Reduction Algorithms. *Theoretical Computer Science*, 53, 12 1987.

[Sch90] Claus-Peter Schnorr. Efficient identification and signatures for smart cards. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 239–252. Springer, Heidelberg, August 1990.

[Sch00] Oliver Schirokauer. Using Number Fields to Compute Logarithms in Finite Fields. *Mathematics of Computation*, 69:1267–1283, 2000.

[Sch05] Oliver Schirokauer. Virtual logarithms. *Journal of Algorithms*, 57:140–147, 2005.

[Sch13] Michael Schneider. Sieving for shortest vectors in ideal lattices. In Amr Youssef, Abderrahmane Nitaj, and Aboul Ella Hassanien, editors, *AFRICACRYPT 13*, volume 7918 of *LNCS*, pages 375–391. Springer, Heidelberg, June 2013.

[SCNS16] Shweta Shinde, Zheng Leong Chua, Viswesh Narayanan, and Prateek Saxena. Preventing Page Faults from Telling Your Secrets. In *AsiaCCS*, pages 317–328. ACM, 2016.

[SE94] Claus-Peter Schnorr and Martin Euchner. Lattice Basis Reduction: Improved Practical Algorithms and Solving Subset Sum Problems. *Math. Program.*, 66(2):181–199, September 1994.

[Sha71] Daniel Shanks. Class number, a theory of factorization, and genera. In Donald J. Lewis, editor, *1969 Number theory institute*, volume 20 of *Proc. Sympos. Pure Math*, pages 415–440. Amer. Math. Soc., 1971.

[Sha82] Adi Shamir. A Polynomial Time Algorithm for Breaking the Basic Merkle-Hellman Cryptosystem. In *23rd Annual Symposium on Foundations of Computer Science (sfcs 1982)*, pages 145–152, 1982.

[Sho97] Victor Shoup. Lower bounds for discrete logarithms and related problems. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 256–266. Springer, Heidelberg, May 1997.

[SP17] Raoul Strackx and Frank Piessens. The Heisenberg Defense: Proactively Defending SGX Enclaves against Page-Table-Based Side-Channel Attacks. *arXiv preprint arXiv:1712.08519*, 2017.

[SPLI06] Junhyuk Song, Radha Poovendran, Jicheol Lee, and Tetsu Iwata. The AES-CMAC Algorithm. RFC 4493, June 2006.

[SS16a] Palash Sarkar and Shashank Singh. Fine tuning the function field sieve algorithm for the medium prime case. *IEEE Transactions on Information Theory*, 62:2233–2253, 2016.

[SS16b] Palash Sarkar and Shashank Singh. A general polynomial selection method and new asymptotic complexities for the tower number field sieve algorithm. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part I*, volume 10031 of *LNCS*, pages 37–62. Springer, Heidelberg, December 2016.

[SS16c] Palash Sarkar and Shashank Singh. New complexity trade-offs for the (multiple) number field sieve algorithm in non-prime fields. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part I*, volume 9665 of *LNCS*, pages 429–458. Springer, Heidelberg, May 2016.

[SS19] Palash Sarkar and Shashank Singh. A unified polynomial selection method for the (tower) number field sieve algorithm. *Advances in Mathematics of Communications*, 13:435–455, 2019.

[SSTX09] Damien Stehlé, Ron Steinfeld, Keisuke Tanaka, and Keita Xagawa. Efficient public key encryption based on ideal lattices. In Mitsuru Matsui, editor, *ASIACRYPT 2009*, volume 5912 of *LNCS*, pages 617–635. Springer, Heidelberg, December 2009.

[SWG$^+$17] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *DIMVA*, pages 3–24, 2017.

[Tes00] Edlyn Teske. On Random Walks For Pollard's Rho Method. *Mathematics of Computation*, 70:809–825, 2000.

[The16] The FPLLL development team. FPLLL, a lattice reduction library. Found at https://github.com/fplll/fplll, 2016.

[Tho02] Emmanuel Thomé. Subquadratic computation of vector generating polynomials and improvement of the block Wiedemann algorithm. *Journal of Symbolic Computation*, 33(5):757–775, 2002.

[TOS10] Eran Tromer, Dag Arne Osvik, and Adi Shamir. Efficient Cache Attacks on AES, and Countermeasures. *J. Cryptology*, 23:37–71, 07 2010.

[TSS$^+$03] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers with cache. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *CHES 2003*, volume 2779 of *LNCS*, pages 62–76. Springer, Heidelberg, September 2003.

[TTA18] Akira Takahashi, Mehdi Tibouchi, and Masayuki Abe. New Bleichenbacher records: Fault attacks on qDSA signatures. *IACR TCHES*, 2018(3):331–371, 2018. .

[TTMH02] Yukiyasu Tsunoo, Etsuko Tsujihara, Kazuhiko Minematsu, and Hiroshi Hiyauchi. Cryptanalysis of Block Ciphers Implemented on Computers with Cache. In *International Symposium on Information Theory and Its Applications*, Xi'an, CN, October 2002.

[Van92] Scott Vanstone. Responses to NIST's proposals, 1992.

[VBWK$^+$17] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. Telling your Secrets without Page Faults: Stealthy Page Table-based Attacks on Enclaved Execution. In *USENIX Security Symposium*. USENIX Association, 2017.

[vEB81] Peter van Emde Boas. Another NP-complete Problem and the Complexity of Computing Short Vectors in a Lattice. *Tecnical Report, Department of Mathmatics, University of Amsterdam*, 1981.

[vSY15] Joop van de Pol, Nigel P. Smart, and Yuval Yarom. Just a little bit more. In Kaisa Nyberg, editor, *CT-RSA 2015*, volume 9048 of *LNCS*, pages 3–21. Springer, Heidelberg, April 2015.

[WF17] Wenbo Wang and Shuqin Fan. Attacking OpenSSL ECDSA with a small amount of side-channel information. *Science China Information Sciences*, 61(3):032105, 2017.

[Wie86] Douglas H. Wiedemann. Solving sparse linear equations over finite fields. *IEEE Trans. Information Theory*, 32:54–62, 1986.

[WLTB11] Xiaoyun Wang, Mingjie Liu, Chengliang Tian, and Jingguo Bi. Improved Nguyen-Vidick Heuristic Sieve Algorithm for Shortest Vector Problem. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 1–9, New York, NY, USA, 2011. Association for Computing Machinery.

[WWB+17] Shuai Wang, Wenhao Wang, Qinkun Bao, Pei Wang, XiaoFeng Wang, and Dinghao Wu. Binary Code Retrofitting and Hardening Using SGX. In *FEAST'17*, pages 43–49. ACM, 2017.

[XCP15] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *IEEE Symposium on Security and Privacy (SP)*, pages 640–656. IEEE, 2015.

[XLCZ17] Yuan Xiao, Mengyuan Li, Sanchuan Chen, and Yinqian Zhang. STACCO: Differentially Analyzing Side-Channel Traces for Detecting SSL/TLS Vulnerabilities in Secure Enclaves. In *CCS*, pages 859–874, 2017.

[YB14] Yuval Yarom and Naomi Benger. Recovering OpenSSL ECDSA nonces using the FLUSH+RELOAD cache side-channel attack. Cryptology ePrint Archive, Report 2014/140, 2014. .

[YF14] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014*, pages 719–732. USENIX Association, August 2014.

[YGH16] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *CHES 2016*, volume 9813 of *LNCS*, pages 346–367. Springer, Heidelberg, August 2016.

[YGL+15] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. Mapping the intel last-level cache. Cryptology ePrint Archive, Report 2015/905, 2015. .

[YKSA15] Younis A. Younis, Kashif Kifayat, Qi Shi, and Bob Askwith. A New Prime and Probe Cache Side-Channel Attack for Cloud Computing. In *2015 IEEE International Conference on Computer and Information Technology; Ubiquitous Computing and Communications; Dependable, Autonomic and Secure Computing; Pervasive Intelligence and Computing*, pages 1718–1724, 2015.

[ZPH14] Feng Zhang, Yanbin Pan, and Gengran Hu. A three-level sieve algorithm for the shortest vector problem. In Tanja Lange, Kristin Lauter, and Petr Lisoněk, editors, *Selected Areas in Cryptography – SAC 2013*, pages 29–47, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

Résumé en Français

La cryptographie à clé publique

La cryptographie s'intéresse au problème de l'échange de messages chiffrés, c'est-à-dire inintelligibles, que seul un récepteur légitime peut déchiffrer, donc lire. Afin d'assurer une transmission sécurisée de ces messages, une clé secrète est généralement partagée entre l'expéditeur et le destinataire. Cela pose la difficulté importante d'échanger de manière sécurisée la clé secrète mentionnée ci-dessus.

Au début des années 1970, Merkle a commencé à s'écarter de ce concept de clé partagée et ses idées publiées en 1978 [Mer78] ont été reprises dans l'article fondateur de Diffie et Hellman [DH76], *New directions in Cryptography*. Dans leur article, Diffie et Hellman formalisent la notion de cryptographie à clé publique où deux clés mathématiquement liées sont générées et utilisées : une clé publique et une clé secrète. Un message est ensuite chiffré à l'aide de la clé publique du récepteur. Ce dernier sera alors le seul capable de déchiffrer le message à l'aide de sa clé secrète correspondante.

Les cryptosystèmes à clé publique, également connus sous le nom de protocoles asymétriques, sont tous construits en tenant compte de la notion de fonction à sens unique. Cette dernière correspond à une fonction qui est facile à calculer pour toute entrée donnée mais difficile à inverser. Cette notion correspond bien aux exigences d'un protocole asymétrique. En effet, pour qu'un protocole soit sûr et efficace, le déchiffrement d'un message sans la clé secrète doit être proche de l'impossible, alors que le chiffrement d'un message et le déchiffrement avec la clé secrète doivent être faciles, c'est-à-dire réalisés uniquement avec des opérations simples.

C'est naturellement vers des problèmes mathématiques difficiles que les cryptographes se sont tournés pour trouver des primitives appropriées pour leurs protocoles. Historiquement, deux candidats ont émergé : la multiplication de deux nombres premiers et l'exponentiation modulaire. L'inverse de ces fonctions consiste à factoriser un nombre entier et à calculer un logarithme discret. La difficulté de la factorisation est au cœur du cryptosystème RSA, bien connu et déployé [RSA78]. Dans cette thèse, nous nous concentrerons sur le second candidat : l'exponentiation modulaire et son opération inverse, le calcul d'un logarithme discret.

Exponentiation modulaire et logarithme discret

L'exponentiation modulaire consiste à calculer le reste d'une division euclidienne d'un entier g élevé à une puissance x par un entier positif N, en calculant $g^x \pmod N$. Cette opération est fondamentale dans la théorie computationnelle des nombres où elle se retrouve par exemple dans le petit théorème de Fermat utilisé pour le test de primalité. L'exponentiation modulaire est également largement utilisée en cryptographie à clé publique, où les éléments de groupes tels que les groupes multiplicatifs des corps finis, $\mathbb{Z}/N\mathbb{Z}$ ou le groupe des points rationnels des courbes elliptiques, sont souvent élevés à de grandes puissances.

Pour des raisons pratiques, les opérations utilisées dans les protocoles cryptographiques doivent être faciles et efficaces à réaliser. L'attrait de l'exponentiation modulaire pour la cryptographie provient en partie de la simplicité de son calcul. En effet, le calcul d'une exponentiation modulaire peut se résumer à multiplier g par lui-même $x-1$ fois, puis à prendre le résultat modulo N. Cependant, cette méthode serait très inefficace dans un contexte cryptographique en raison de la taille des nombres impliqués. Par conséquent, afin de construire des protocoles cryptographiques pratiques qui utilisent l'exponentiation modulaire, l'opération doit être effectuée de manière efficace.

L'efficacité des algorithmes qui calculent l'exponentiation modulaire dépend de divers paramètres tels

que le groupe considéré, la représentation de l'exposant ou le matériel utilisé. Comme l'exponentiation modulaire est présente dans de nombreux protocoles, et qu'elle est souvent l'opération la plus coûteuse du protocole, au fil des années, de nombreux algorithmes optimisés se sont accumulés pour améliorer son calcul.

La plupart des algorithmes qui tentent d'optimiser l'exponentiation modulaire visent à réduire le nombre de multiplications nécessaires. L'une des méthodes les plus anciennes et les plus simples est l'algorithme du carré et de la multiplication (Square-and-Multiply, en anglais). Cette méthode est apparue il y a plus de 2000 ans en Inde [Knu97]. L'algorithme opère bit par bit sur les bits de l'exposant et n'effectue une opération de multiplication que si ce bit de l'exposant est 1.

Des algorithmes d'exponentiation modulaire plus sophistiqués font appel à différentes représentations de l'exposant, comme la forme non adjacente (NAF) ou sa variante à fenêtre (wNAF). Nous renvoyons le lecteur à [Gor98] pour une étude des méthodes d'exponentiation rapide.

Bien que de nombreux efforts aient été déployés pour optimiser les algorithmes d'exponentiation modulaire, ces optimisations ont fait apparaître des vulnérabilités exploitables. En effet, les opérations effectuées dans ces algorithmes sont souvent dépendantes des valeurs binaires de l'exposant. L'exécution du code peut alors générer des fuites observables à partir desquelles des informations peuvent être déduites sur l'exposant. Le caractère spécifique des informations divulguées dépend des détails de la mise en œuvre de l'algorithme et souvent du matériel lui-même. Les attaques par canaux auxiliaires et en particulier les attaques par cache sont les principales menaces à prendre en compte lors de l'utilisation d'un algorithme d'exponentiation modulaire rapide pour un protocole.

L'opération inverse de l'exponentiation modulaire est le calcul d'un logarithme discret. L'étude des logarithmes discrets et des algorithmes associés précède leur utilisation en cryptographie. En effet, dès le 19e siècle, les logarithmes de Zech sont utilisés pour accélérer les opérations arithmétiques dans les corps finis.

En cryptographie, le protocole Diffie-Hellman datant de la fin des années 70 a marqué un tournant dans l'étude des logarithmes discrets. L'utilisation plus récente des logarithmes discrets dans les protocoles basés sur les couplages, qui a débuté au début des années 2000, a relancé l'intérêt pour le sujet. Concrètement, un logarithme discret est défini comme suit.

> **Définition 27** (Logarithme discret)**.** *Étant donné un groupe cyclique fini G d'ordre n, un générateur $g \in G$ et un élément $h \in G$, le logarithme discret de h en base g est l'élément $x \in [0, n[$ tel que $g^x = h$.*

Cette définition pose le problème suivant.

> **Définition 28** (Le problème du logarithme discret (DLP))**.** *Étant donné un groupe cyclique fini G d'ordre n, un générateur $g \in G$, et un élément $h \in G$, trouver x tel que $g^x = h$.*

Ce problème est considéré comme difficile pour la plupart des groupes G et constitue donc un candidat prometteur pour la cryptographie à clé publique.

> **Question 90.** *Où peut-on trouver des logarithmes discrets et des exponentiations modulaires en cryptographie ?*

Des cryptosystèmes largement déployés, tels que le protocole d'échange de clés Diffie-Hellman, le protocole de chiffrement d'ElGamal ou les protocoles de signature tels que (EC)DSA basent leur sécurité sur des hypothèse liées à la difficulté du problème du logarithme discret. Nous décrivons certains de ces protocoles.

Le protocole d'échange de clés de Diffie-Hellman [DH76]. Le protocole est assez simple. Deux entités, Alice et Bob, souhaitent communiquer et pour ce faire, elles doivent d'abord se mettre d'accord sur une clé secrète. Elles commencent par choisir un groupe G d'ordre n et un générateur g de ce groupe, qui sont maintenant des paramètres publiques. Alice choisit ensuite un élément aléatoire $a \in [0, n[$ et envoie l'élément de groupe g^a à Bob via un canal publique. De même, Bob choisit $b \in [0, n[$ et envoie g^b à

Alice. Les quantités, g, g^a, g^b sont toutes publiques. Alice et Bob peuvent maintenant tous deux calculer la quantité

$$s = (g^a)^b = (g^b)^a,$$

qui constitue le secret partagé utilisé pour les communications futures.

Un attaquant qui souhaite intercepter une conversation entre Alice et Bob doit récupérer la valeur secrète s. La récupération de g^{ab} à partir de g, g^a, g^b est connue comme le problème calculatoire de Diffie-Hellman, qui est étroitement lié au calcul des logarithmes discrets [MW96]. Le protocole d'échange de clés Diffie-Hellman est devenu une norme en 2003 (ANSI X9.42) et est utilisé dans des protocoles largement déployés tels que HTTPS, SSH/TLS.

Le protocol de chiffrement d'ElGamal [ElG85]. Le protocole de chiffrement d'ElGamal est étroitement lié à l'échange de clés Diffie-Hellman. Alice veut envoyer un message chiffré à Bob. Comme pour tout protocole de chiffrement à clé publique, Bob doit générer une paire de clés, une publique et une privée, qui sont mathématiquement liées. Pour ce faire, il choisit un groupe G d'ordre n ainsi qu'un générateur g de G. La clé privée de Bob sera un élément $b \in [0, n[$ choisi au hasard et connu seulement par Bob. La clé publique est constituée des éléments (G, g, g^b).

Afin de chiffrer un message donné m vu comme un élément de G, Alice utilisera la clé publique de Bob. Alice commence par choisir un élément aléatoire $r \in [0, n[$ et calcule la quantité $c = m \cdot (g^b)^r$. Alice calcule également $c' = g^r$. Le texte chiffré est donc composé des deux quantités (c, c') qu'Alice envoie à Bob.

Une fois que Bob reçoit le texte chiffré, il peut déchiffrer le message m en utilisant sa clé privée b. En effet, Bob calcule $c(c'^b)^{-1} = m$.

Un attaquant qui intercepte le texte chiffré (c, c') et souhaite le déchiffrer devrait résoudre le problème calculatoire de Diffie-Hellman. Le schéma de chiffrement ElGamal est largement utilisé pour les systèmes de vote [BW14].

Le protocole de signature d'ElGamal [ElG85]. Considérons $G = (\mathbb{Z}/p\mathbb{Z})^*$ pour un nombre premier p. Alice veut envoyer un message m à Bob et en plus elle veut signer le message afin de l'authentifier. Comme précédemment, elle possède une paire de clés secrètes/publiques (a, g^a) où g est un générateur du groupe G considéré. Pour générer sa signature, Alice choisit un entier aléatoire $k \in [0, p-1]$ où p est l'ordre premier de G, et tel que k et $p-1$ sont premiers entre-eux. Elle calcule alors les deux quantités $r = g^k \pmod{p}$ et $s = (m - ar)k^{-1} \pmod{p-1}$. Sa signature est la paire (r, s) et elle l'envoie à Bob avec le message m.

Si Bob veut vérifier la validité du message, il vérifie la signature d'Alice en utilisant sa clé publique. D'après la génération de la signature, nous savons que $m = ar + sk \pmod{p-1}$. Puisque Bob connaît la clé publique d'Alice, g^a, nous pouvons comparer les quantités g^m et $g^{ar}g^{sk}$ modulo p.

D'autres schémas de signature tels que l'algorithme de signature numérique (DSA) de 1991 (spécifications dans FIPS 186-4 [NIS13]), une variante du schéma de signature ElGamal, et sa variante utilisant les courbes elliptiques ECDSA [JMV01], sont également des protocoles normalisés basés sur la difficulté de DLP.

Protocoles de couplage. La cryptographie basée sur les couplages est au cœur de nombreux produits de sécurité qui sont mis sur le marché et la recherche de primitives efficaces les utilisant est très active. C'est le cas notamment dans le domaine du zero-knowledge avec les applications de Zk-SNARKs aux smart contracts.

Les preuves zero-knowledge permettent à un vérificateur de certifier qu'un prouveur a connaissance d'un secret sans révéler d'informations sur le secret lui-même. Les Zk-SNARKs, Zero-knowledge Succinct Non-interactive Argument of Knowledge, sont des exemples de protocoles largement déployés dans les smart contracts qui utilisent des preuves zero-knowledge. Beaucoup de ces protocoles utilisent des couplages dans leurs constructions. L'évaluation de la sécurité de ces schémas est donc fondamentale. Concrètement, un couplage cryptographique est défini comme suit.

Definition 29 (Couplage cryptograhique). *Considérons les groupes abéliens finis* $\mathbb{G}_1$, $\mathbb{G}_2$, *et* $\mathbb{G}_T$ *d'ordre n. Un couplage cryptographique est une application*

$$e : \mathbb{G}_1 \times \mathbb{G}_2 \to \mathbb{G}_T,$$

qui est

- *bilinéaire, c'est-à-dire que pour tout* $a, b \in [0, n[$, $P \in \mathbb{G}_1$ *et* $Q \in \mathbb{G}_2$ *on a*

$$e(aP, bQ) = e(P, Q)^{ab},$$

- *non-dégénérée, c'est-à-dire que pour tout* $P \in \mathbb{G}_1$, *il existe* $Q \in \mathbb{G}_2$ *tel que* $e(P, Q) \neq 1$ *et pour tout* $Q \in \mathbb{G}_2$ *il existe* $P \in \mathbb{G}_1$ *tel que* $e(P, Q) \neq 1$,

- *et calculable en temps polynomial dans la taille de l'entrée.*

La sécurité des protocoles basés sur les couplages repose sur la difficulté du problème du logarithme discret. En effet, au début des années 2000, la cryptographie basée sur les couplages a introduit de nouveaux schémas tels que le chiffrement basé sur l'identité [BF01], la signature basée sur l'identité [CC03], la signature courte [BLS01] ou les schémas de signature aveugle utilisés par exemple dans les enclaves SGX d'Intel (voir le chapitre 7) dont la sécurité est basée sur des hypothèses liées aux couplages qui deviennent fausses si le DLP est cassé. Pour construire un protocole sécurisé basé sur un couplage, on doit donc supposer que les DLP dans les trois groupes $\mathbb{G}_1, \mathbb{G}_2, \mathbb{G}_T$ sont difficiles.

Une question naturelle découle des descriptions ci-dessus des schémas et du problème du logarithme discret.

Question 91. *Quel groupe G doit-on considérer?*

L'évaluation de la difficulté de DLP est depuis longtemps un domaine de recherche actif. De nombreux algorithmes pour résoudre DLP sont apparus au fil des années. Ces algorithmes varient dans leur construction et leur complexité dépend du groupe G considéré. La cryptographie a renouvelé l'intérêt pour le problème du logarithme discret sur des groupes spécifiques. En pratique, le groupe G dans la définition du problème du logarithme discret est choisi comme étant soit le groupe multiplicatif d'un corps fini $\mathbb{F}_{p^n}$ ou le groupe des points rationnels sur une courbe elliptique $\mathcal{E}$ définie sur un corps fini.

La cryptographie basée sur les couplages illustre la nécessité de considérer le problème de logarithme discret à la fois sur les corps finis et sur les courbes elliptiques. En effet, les groupes considérés pour un couplage cryptographique sont généralement $\mathbb{G}_1$, un sous-groupe de $\mathcal{E}(\mathbb{F}_p)$, le groupe des points d'une courbe elliptique $\mathcal{E}$ définie sur le corps premier $\mathbb{F}_p$, $\mathbb{G}_2$, un autre sous-groupe de $\mathcal{E}(\mathbb{F}_{p^n})$ où l'on considère une extension de corps et $\mathbb{G}_T$ un sous-groupe multiplicatif de ce même corps fini $\mathbb{F}_{p^n}$.

Il est intéressant de noter que le groupe de points rationnels sur une courbe elliptique $\mathcal{E}$ définie sur un corps fini ne fournit aucune représentation utile pour accélérer le calcul d'un logarithme discret. Par conséquent, les meilleurs algorithmes connus pour résoudre DLP sur ce groupe sont les algorithmes génériques avec une complexité en racine carrée.

D'autre part, la prise en compte du problème du logarithme discret dans les corps finis $\mathbb{F}_{p^n}$ a permis d'améliorer considérablement l'efficacité des algorithmes qui le résolvent. La nature et la complexité de ces algorithmes dépendent des caractéristiques du corps fini et plus précisément de la relation entre la caractéristique p et le degré d'extension n. Les algorithmes les plus efficaces pour résoudre DLP dans les corps finis proviennent de la famille des algorithmes de calcul d'indices. Parmi ces algorithmes, nous avons le Function Field Sieve (FFS), le crible algébrique (Number Field Sieve en anglais, NFS) et ses nombreuses variantes, ainsi que les algorithmes plus récents en temps Quasi-Polynomial (QP). Une vue d'ensemble de tous ces algorithmes fait l'objet du Chapitre 1.

En 1994, Shor a introduit un algorithme quantique en temps polynomial pour calculer les logarithmes discrets. Cela implique qu'aucun protocole s'appuyant sur la difficulté de DLP ne serait sûr en présence

d'ordinateurs quantiques, quel que soit le groupe considéré. Cependant, à l'heure actuelle, il n'existe pas d'ordinateurs quantiques capables d'effectuer des calculs à grande échelle, bien que des progrès impressionnants aient été réalisés ces dernières années (voir [AAB+19] pour une machine récente de 53 qubits). Par conséquent, nous limiterons cette thèse à la configuration classique.

Un autre candidat: les réseaux euclidiens

Au cours des dernières décennies, motivés par la menace imminente des ordinateurs quantiques accessibles, de nouveaux candidats prometteurs sont apparus pour construire des protocoles à clé publique : réseaux, isogénies, codes correcteurs d'erreurs, polynômes multivariés et fonctions de hachage. Nous nous concentrons ici sur les réseaux.

L'étude des réseaux en mathématiques a commencé dès le 18e siècle. Des mathématiciens tels que Gauss, Lagrange et Minkowski ont étudié les réseaux dans le contexte de la géométrie des nombres et de la géométrie convexe, plus particulièrement pour la théorie de la réduction des formes quadratiques qui a ensuite conduit au célèbre algorithme de Gauss. Il faut attendre les années 1980 pour que les réseaux soient étudiés d'un point de vue informatique et utilisés dans des domaines plus proches de l'informatique tels que l'optimisation combinatoire et la cryptographie.

En cryptographie, les réseaux sont utilisés pour la première fois pour casser des cryptosystèmes. Des algorithmes tels que l'algorithme de Lenstra, Lenstra, Lovász (LLL) [LLL82] sont développés pour donner des solutions approximatives aux problèmes de réseaux difficiles et sont largement utilisés pour la cryptanalyse. En 1982, Shamir [Sha82] a utilisé l'algorithme LLL pour casser le système de chiffrement de Merkle-Hellman. En 1996, Coppersmith [Cop96b, Cop96a] a proposé une méthode permettant de factoriser un entier n lorsque certains bits des facteurs de n sont connus, en utilisant des algorithmes de réduction de base de réseaux tels que LLL, affectant ainsi la sécurité du cryptosystème RSA.

L'utilisation de réseaux pour la conception de protocoles cryptographiques n'a commencé qu'en 1996 avec les travaux d'Ajtai [Ajt96]. Les premiers cryptosystèmes à utiliser des réseaux comme blocs de construction sont les cryptosystèmes Ajtai-Dwork [AD97] et NTRU [HPS98] à la fin des années 90. Aujourd'hui, la cryptographie basée sur les réseaux est un domaine de recherche important et de nombreux protocoles basés sur les réseaux sont candidats à la compétition post-quantique du NIST. Dans cette thèse, nous ne considérerons pas les protocoles construits sur la difficulté des problèmes liés aux réseaux. Cependant, nous utiliserons les techniques de réseaux dans deux configurations différentes :

- nous utilisons des algorithmes de réduction de réseaux pour produire des polynômes à petits coefficients, voir Chapitre 3 ou pour trouver les petites racines de polynômes modulaires, voir Partie III.

- nous utilisons des algorithmes d'énumération dand les réseaux, en particulier une adaptation de l'algorithme de Schnorr-Euchner, pour accélérer la recherche de relations algébriques dans le contexte des calculs de logarithmes discrets, voir le chapitre 4.

Des informations préliminaires sur les réseaux et les algorithmes associés sont donc données dans le Chapitre 2.

Remark 26. *Dans cette thèse, nous parlerons de crible pour les réseaux et de cible algébrique. Ces deux notions ne désignent pas la même chose ! Le crible algébrique fait généralement référence à une étape de NFS, tandis que le crible pour les réseaux, tel que le Gauss Sieve, fait référence à un algorithme concernant les réseaux qui trouve des vecteurs courts. Le crible pour les réseaux peut également être trouvé dans le contexte de NFS. Nous précisons quel crible est considéré lorsque le contexte n'est pas clair.*

Contributions

L'objectif de cette thèse est de répondre à la question suivante.

Question 92. *Comment évaluer la sécurité des protocoles dans lesquels une exponentiation modulaire impliquant un secret est effectuée ?*

La réponse à cette question se divise en deux points.

1. La résolution du problème du logarithme discret donne un accès direct à l'exposant, donc au secret. Ainsi, nous voulons estimer la difficulté de DLP dans les groupes considérés par les protocoles.

2. L'étude des vulnérabilités d'implémentation pendant l'exponentiation rapide peut également conduire à l'exposant secret. Ainsi, nous voulons également évaluer et étudier les attaques rendues possibles grâce aux informations fuitées par des canaux auxiliaires.

Ces deux points vont façonner la structure de cette thèse.

Estimation de la difficulté de DLP dans les corps finis

Une façon d'estimer la sécurité des protocoles basés sur la difficulté du problème du logarithme discret est d'étudier directement la complexité des algorithmes qui résolvent ce dernier. Comme nous l'avons mentionné plus haut, cela dépend fortement du groupe considéré. Dans cette thèse, nous nous concentrerons sur l'estimation de la difficulté de DLP pour des corps finis spécifiques.

> **Question 93.** *Sur quels corps finis nous concentrons-nous et pourquoi ?*

Les corps finis $\mathbb{F}_{p^n}$ sont généralement séparés en trois familles appelées petite, moyenne et grande caractéristique, en fonction de la relation entre la caractéristique p et le degré d'extension n du corps fini. À ce jour, les algorithmes connus les plus rapides pour résoudre DLP sont les algorithmes en temps quasi-polynomial pour les corps finis de petite caractéristique [BGJT14, KW19].

Cependant, dans cette thèse, nous nous intéresserons aux corps finis compris entre la frontière entre petite et moyenne caractéristique et la grande caractéristique. En effet, comme ces familles ne fournissent pas l'algorithme connu le plus rapide pour résoudre DLP, elles concernent la plupart des corps finis utilisés en pratique, par exemple dans les protocoles basés sur les couplage. Par conséquent, l'estimation de la difficulté du DLP pour ces familles a un impact significatif sur notre compréhension de la sécurité des protocoles largement déployés.

Notre première motivation concerne la sécurité des protocoles basés sur les couplages. Si nous voulons qu'un couplage soit sûr, nous voulons équilibrer la complexité de l'algorithme en racine carrée qui calcule les logarithmes discrets dans le sous-groupe pertinent de la courbe elliptique considérée, et la complexité de l'algorithme le plus rapide qui résout DLP dans le corps fini. Ceci nous a amené à étudier les algorithmes de la famille du calcul d'indice mentionnés ci-dessus à la frontière entre les corps finis de petite caractéristique et ceux de caractéristique moyenne. La complexité asymptotique de ces algorithmes à ce cas frontière était, jusqu'à cette thèse, inexistante dans la littérature. Cette étude nous a également permis de fournir des points d'intersection précis entre ces nombreuses complexités. Ce sera l'objet du Chapitre 3, illustré par la figure 1. Grâce à cette analyse, nous avons finalement pu fournir des informations supplémentaires sur les paramètres de sécurité des protocoles basés sur les couplages. Plus précisément, ce chapitre répond à la question suivante.

> **Question 94.** *Asymptotiquement, quel corps fini $\mathbb{F}_{p^n}$ devrait être considéré afin d'obtenir le plus haut niveau de sécurité lors de la construction d'un couplage ?*

Dans ce chapitre, nous donnons des valeurs optimales pour la caractéristique p et le degré d'extension n, en prenant également en compte la valeur dite ρ des constructions de couplages. Fait surprenant, nous avons pu distinguer quelques caractéristiques spéciales qui sont asymptotiquement aussi sûres que les caractéristiques de même taille mais sans forme spéciale. L'article suivant résume nos résultats.

1. **Asymptotic complexities of discrete logarithm algorithms in pairing-relevant finite fields**, avec Pierrick Gaudry et Cécile Pierrot, publié dans les actes de la conférence Crypto 2020.

Une autre façon d'obtenir de meilleures estimations de sécurité consiste à réaliser des expériences à grande échelle avec des variantes du Number Field Sieve. En effet, l'algorithme Number Field Sieve a donné lieu à de nombreuses variantes, chacune tentant de réduire la complexité asymptotique de l'algorithme original. L'une de ces variantes est le Tower Number Field Sieve (TNFS). Ce dernier exploite la structure algébrique des tours de corps de nombres. Malgré le fait qu'en théorie la variante est plus
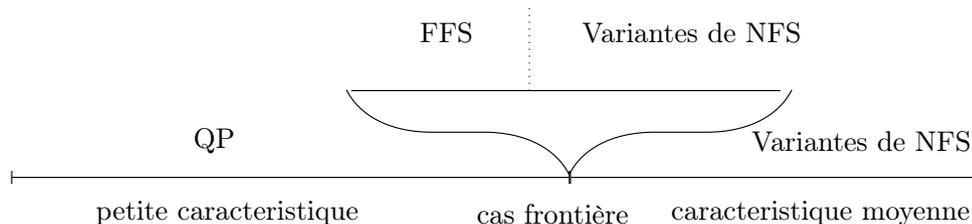
Figure 1: Représentation des corps finis et algorithmes associés étudiés dans le Chapitre 3.

que prometteuse, aucune implémentation et donc aucun calcul record n'avait été fait en utilisant TNFS, jusqu'à cette thèse.

Un obstacle majeur à une mise en œuvre efficace de TNFS est la collection de relations algébriques où des équations entre de petits éléments de corps de nombres doivent être trouvées. Le cas de TNFS est plus complexe que celui de NFS car cette collecte de relations se produit en dimension supérieure à 2. Cela nécessite la construction de nouveaux algorithmes de crible qui restent efficaces lorsque la dimension augmente. Dans le Chapitre 4, nous surmontons cette difficulté en considérant un algorithme d'énumération sur les réseaux que nous adaptons à ce contexte spécifique. Nous considérons également une nouvelle zone de crible, une sphère de haute dimension, alors que les algorithmes de crible précédents (pour les dimensions 2 et 3) considéraient un hyper-parallélépipède.

Cela nous a permis d'effectuer le premier calcul record d'un logarithme discret avec TNFS dans un corps fini de 521 bits $\mathbb{F}_{p^6}$. Le corps fini cible $\mathbb{F}_{p^6}$ choisi est de la même forme que les corps finis utilisés dans les récentes preuves zéro knowledege de certaines blockchains. Ce calcul record a été annoncé en février 2021.

2. **Discrete logarithm in GF(p^6) with Tower NFS** avec Pierrick Gaudry et Cécile Pierrot, annoncé dans liste de diffusion de Théorie des Nombres. Article correspondant en cours de soumission.

Les détails de l'implémentation et du calcul sont donnés dans le Chapitre 5. Comme on peut le voir dans le Tableau 1, notre algorithme est beaucoup plus rapide que les algorithmes de crible en dimension supérieur à deux existants malgré la plus grande dimension et le plus grand corps fini.

Paramètres	[GGMT17]	[MR21]	Cette thèse
Algorithme	NFS	NFS	TNFS
Taille du corps fini (bits)	422	423	521
Dimension de crible	3	3	6
Temps de crible	201,600	69,120	**23,300**

Table 1: Comparison de l'étape de collecte de relations en heures de calcul sur un coeur avec [GGMT17] et [MR21] pour $\mathbb{F}_{p^6}$.

Ces deux travaux contribuent à estimer la difficulté du DLP dans les corps finis en étudiant les complexités asymptotiques des algorithmes pertinents et en fournissant un calcul record avec TNFS. Les considérations faites sur la sécurité des couplages devraient compléter les estimations pratiques trouvées dans la littérature et, espérons-le, orienter les cryptanalystes vers les bons choix de paramètres. Les performances pratiques de TNFS avec notre nouvel algorithme de crible sont prometteuses et indiquent que des corps finis plus grands pourraient être atteints en un temps raisonnable. En général, les calculs records fournissent des indications supplémentaires sur l'écart entre les tailles de clés recommandées pour les protocoles basés sur DLP et ce qui est faisable sur le plan informatique.

Exploitation des vulnérabilités de l'exponentiation rapide

La sécurité des protocoles déployés ne dépend pas seulement de la difficulté du problème mathématique sous-jacent, mais aussi de l'implémentation des algorithmes concernés.

Les implémentations vulnérables de l'exponentiation modulaire rapide ont souvent été la cible d'attaques par canaux auxiliaires où des informations secrètes sont récupérées en créant des liens observables entre les différentes unités d'exécution du CPU. En particulier, les attaques temporelles exploitent les variations du temps d'exécution qui sont courantes dans les algorithmes d'exponentiation modulaire.

Dans le Chapitre 6, nous présentons un aperçu des techniques connues pour récupérer des clés secrètes à partir d'informations partielles. La fuite d'information, généralement un certain nombre de bits d'un élément secret du protocole, est illustrée dans la Figure 2.

Information partielle (bits)

Exposant secret

Figure 2: Exemple de representation de l'information partielle fuitée par une attaque par canaux auxiliaires.

De nombreuses techniques de récupération de clés secrètes à partir d'informations partielles existent en fonction de la nature de l'information récupérée par l'attaque par canaux auxiliaires et des spécificités de l'algorithme utilisé. Ce chapitre présente les techniques les plus utiles ainsi qu'une classification complète de ce qui est connu pour être efficace pour les scénarios les plus couramment rencontrés dans la pratique. Nous nous concentrons sur les algorithmes largement utilisés qui sont les cibles les plus populaires des attaques, à savoir RSA, (EC)DSA et Diffie-Hellman (ainsi que sa variante avec une courbe elliptique). Nos résultats figurent dans l'article suivant.

3. **Recovering cryptographic keys from partial information, by example**, avec Nadia Heninger. Mis en ligne sur Eprint:Report 2020/1506.

Les techniques présentées dans le Chapitre 6 ont souvent conduit à des attaques réelles sur des protocoles déployés. Dans cette thèse, nous nous concentrons sur deux de ces techniques qui reposent sur des constructions de réseaux : le Hidden Number Problem et le Extended Hidden Number Problem.

Dans le Chapitre 7, nous étudions la sécurité de l'implémentation d'Intel du protocole EPID (Extended Privacy ID), un protocole d'authentification et d'attestation à distance. Nous identifions une faiblesse d'implémentation qui fait fuiter des informations via un canal auxiliaire du cache. Cette fuite d'information nous permet de monter une approche basée sur les réseaux pour résoudre le Hidden Number Problem, que nous adaptons à la preuve zero-knowledge du protocole EPID, étendant ainsi les attaques antérieures sur les systèmes de signature. Ce travail montre qu'un fournisseur d'attestation malveillant peut utiliser l'information divulguée pour briser les garanties de non-liaison d'EPID. Nous fournissons également des preuves expérimentales que l'attaque par réseaux peut toujours réussir même lorsqu'un petit nombre de traces erronées est inclus. Ces résultats sont présentés dans l'article suivant.

4. **CacheQuote: Efficiently Recovering Long-term Secrets of SGX EPID via Cache Attacks**, avec Fergus Dall, Thomas Eisenbarth, Daniel Genkin, Nadia Heninger, Ahmad Moghimi et Yuval Yarom, publié dans le journal IACR Transactions on Cryptographic Hardware and Embedded Systems,Volume 2, 2018.

Nous nous concentrons enfin sur la sécurité du protocole ECDSA lorsque le nonce k utilisé dans l'algorithme de signature comme exposant modulaire est exprimé sous la forme wNAF. Dans le Chapitre 8, nous réétudions la construction du réseau utilisé dans le Extended Hidden Number Problem (EHNP). Nous trouvons la clé secrète avec seulement 3 signatures, atteignant ainsi une limite théorique connue, alors que les meilleures méthodes précédentes nécessitaient au moins 4 signatures en pratique. Étant donné un modèle de fuite spécifique, notre attaque est plus efficace que les attaques précédentes et, dans la plupart des cas, a une meilleure probabilité de succès. Nous fournissons également une première analyse de la résistance aux erreurs de EHNP. Ce travail est décrit dans l'article suivant.

5. **A Tale of Three Signatures: Practical Attack of ECDSA with wNAF**, avec Cécile Pierrot et Rémi Piau, publié dans les actes de la conférence Africacrypt 2020.

En considérant des cibles réelles telles que EPID dans l'architecture d'Intel et l'algorithme ECDSA largement déployé, nous montrons tout au long de ces travaux que même si les bons paramètres sont pris en compte pour que le problème du logarithme discret reste suffisamment difficile à résoudre à des fins cryptographiques, les attaques peuvent provenir d'implémentations vulnérables de l'exponentiation modulaire. Afin d'évaluer réellement la sécurité des protocoles à clé publique déployés, il faut donc considérer simultanément les menaces provenant de la primitive mathématique elle-même et de l'implémentation des algorithmes.

Les contributions et l'organisation de la thèse sont résumées dans la Figure 3.
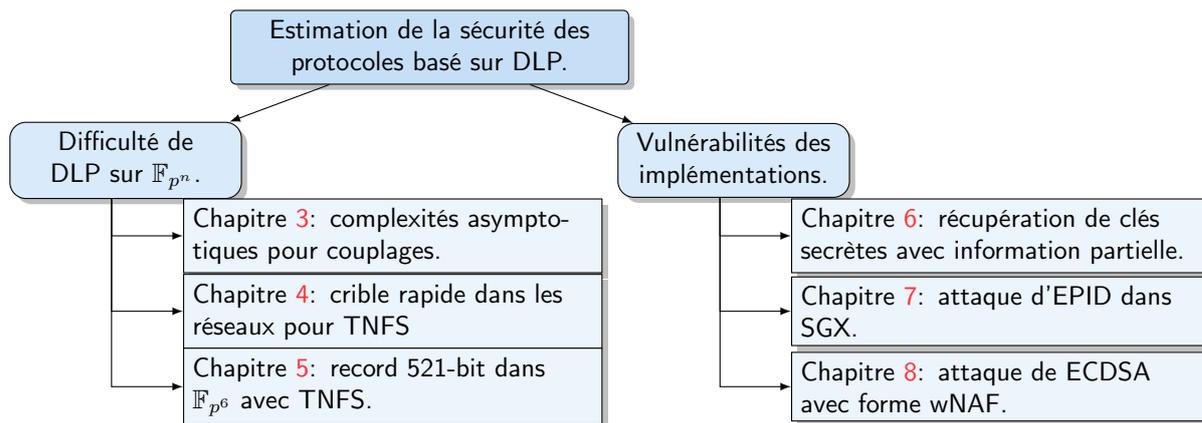

Figure 3: Organisation des contributions de la thèse.

Autre contribution

Overstretched NTRU est une variante de NTRU avec un grand modulo. De récentes attaques de sous-corps et de sous-anneaux de réseaux ont brisé les paramètres suggérés pour plusieurs schémas. Il existe un certain nombre d'affirmations contradictoires dans la littérature sur l'attaque qui présente les meilleures performances. Ces affirmations sont généralement basées sur des expériences plutôt que sur des analyses.

Dans ce travail, nous soutenons que les comparaisons devraient se concentrer sur la dimension du réseau utilisé dans l'attaque. Nous donnons des preuves, à la fois analytiques et expérimentales, que l'attaque par sous-anneaux trouve des vecteurs plus courts et devrait donc réussir avec un réseau de plus petite dimension que l'attaque par sous-corps pour les mêmes paramètres du problème, et également réussir avec un modulo plus petit lorsque la dimension du réseau est fixée.

Comme la thématique de ce travail est en dehors du thème principal de cette thèse, nous ne l'incluons pas dans le manuscrit. L'article suivant résume ces résultats.

6. **Characterizing overstretched NTRU attacks**, avec Nadia Heninger et Barak Shani. Présenté à Mathcrypt 2018, publié dans Journal of Mathematical Cryptology, Volume 14, no. 1, 2020.

Abstract

Public-key cryptosystems are constructed using one-way functions which ensure both the security and the efficiency of the schemes. One of the two main candidates originally considered to construct public-key cryptosystems is modular exponentiation with its hard inverse operation, computing discrete logarithms. In this thesis, we study the security of protocols that make use of modular exponentiation where the exponent is a secret of the protocol. In order to assess the security of such protocols, one can either estimate the hardness of directly solving the discrete logarithm problem (DLP) in the groups considered by the protocols or look at implementation vulnerabilities coming from fast exponentiation algorithms.

One way of estimating the security of protocols based on the hardness of the discrete logarithm problem is to directly study the complexity of the algorithms that solve the latter. In this thesis, we first study the asymptotic complexity of algorithms that solve DLP over finite fields $\mathbb{F}_{p^n}$ precisely of the form where pairings take their values. These algorithms come from the index-calculus family from which the Number Field Sieve (NFS) is an example. This study allows us to draw conclusions on the security of pairing-based protocols. We also propose a first implementation of the variant Tower Number Field Sieve (TNFS) of NFS, which has better asymptotic complexity, along with a record computation of a discrete logarithm in a 521-bit finite field with TNFS. This variant had never been implemented before due to the difficulty of sieving in higher dimensions, *i.e.,* dimensions greater than two.

Finally, the security of deployed protocols not only relies on the hardness of the underlying mathematical problem but also on the implementation of the algorithms involved. Many fast modular exponentiation algorithms have piled up over the years and some implementations have brought forth vulnerabilities that are exploitable by side-channel attacks, in particular cache attacks. The second aspect of this thesis thus considers key recover methods when partial information is recovered from a side channel.

Résumé

Les cryptosystèmes dits à clé publique sont construits à l'aide de fonctions à sens unique qui assurent à la fois la sécurité et l'efficacité des cryptosystèmes. L'un des deux principaux candidats envisagés à l'origine pour construire de tels cryptosystèmes est l'exponentiation modulaire avec son opération inverse, le calcul de logarithmes discrets. Dans cette thèse, nous étudions la sécurité de protocoles qui utilisent des exponentiations modulaires où l'exposant est un secret du protocole. Afin d'évaluer la sécurité de tels protocoles, on peut d'une part estimer la difficulté de résoudre directement le problème du logarithme discret (DLP) dans les groupes considérés par les protocoles, ou examiner les vulnérabilités issues de l'implémentation des algorithmes d'exponentiation rapide.

Une première façon d'estimer la sécurité des protocoles basés sur la difficulté du problème du logarithme discret est d'étudier directement la complexité des algorithmes qui résolvent ce dernier. Dans cette thèse, nous étudions la complexité asymptotique des algorithmes qui résolvent le DLP sur des corps finis $\mathbb{F}_{p^n}$ précisément de la forme où les couplages prennent leurs valeurs.

Nous proposons également une première implémentation et un calcul record d'un logarithme discret dans un corps fini de 521 bits en utilisant l'algorithme Tower Number Field Sieve, une variante de NFS dont la complexité asymptotique est meilleure. Cette variante n'avait jamais été implémentée auparavant en raison de la difficulté du crible algébrique dans des dimensions supérieures à deux.

Enfin, la sécurité des protocoles déployés ne repose pas seulement sur la difficulté du problème mathématique sous-jacent, mais aussi sur l'implémentation des algorithmes considérés. De nombreux algorithmes d'exponentiation modulaire rapide se sont accumulés au fil des ans et certaines implémentations ont fait apparaître des vulnérabilités exploitables par des attaques par canaux auxiliaires. Un second aspect de cette thèse considère donc les principales méthodes pour reconstituer une clé secrète lorsque des informations partielles sont récupérées à partir d'un canal auxiliaire.