Key recovery from partial information

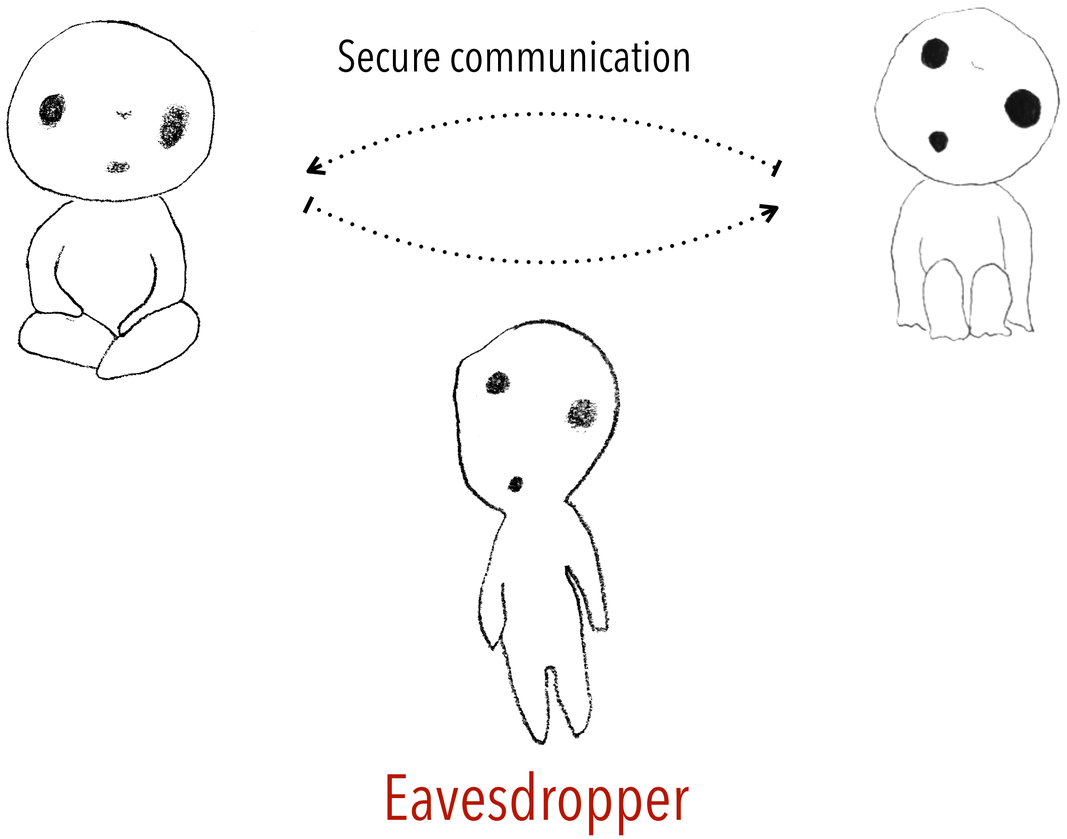
Séminaire de Cryptographie, Rennes

Gabrielle De Micheli, UC San Diego

1er Octobre, 2021

UC San Diego

What? Why? Where? Cryptography …

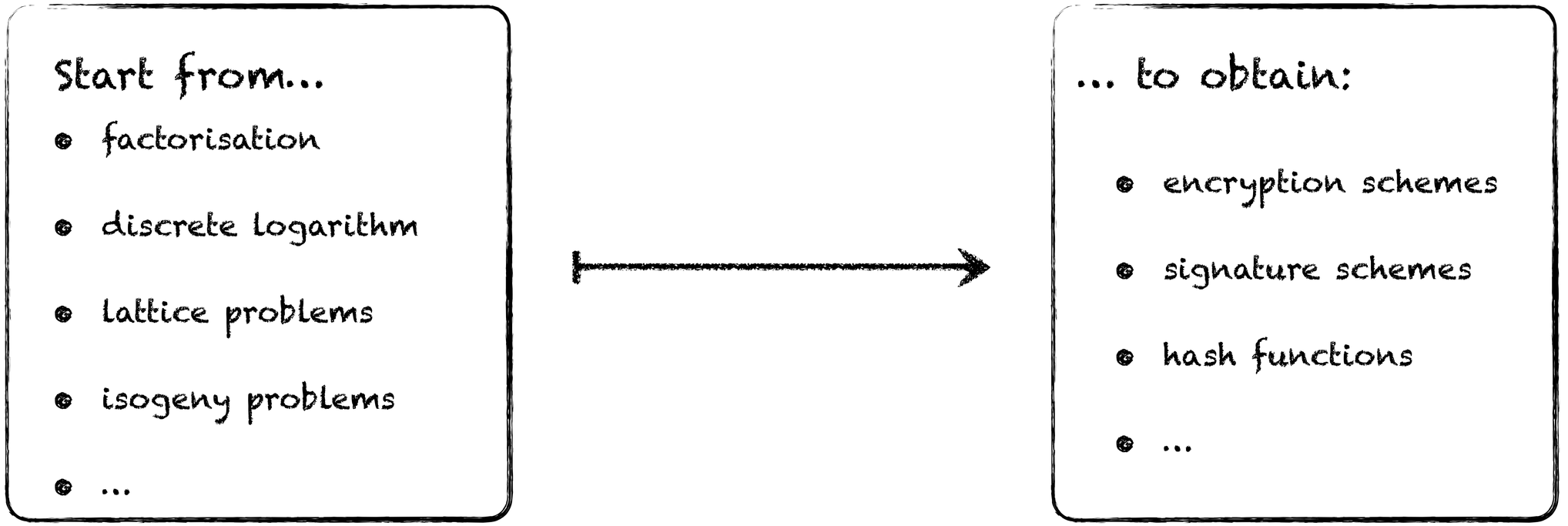


Cryptographic protocols for:

- Confidentiality (encryption schemes)
- Authentication and non-repudiation (signature schemes)
- Integrity and validity of data (hash functions)
- …

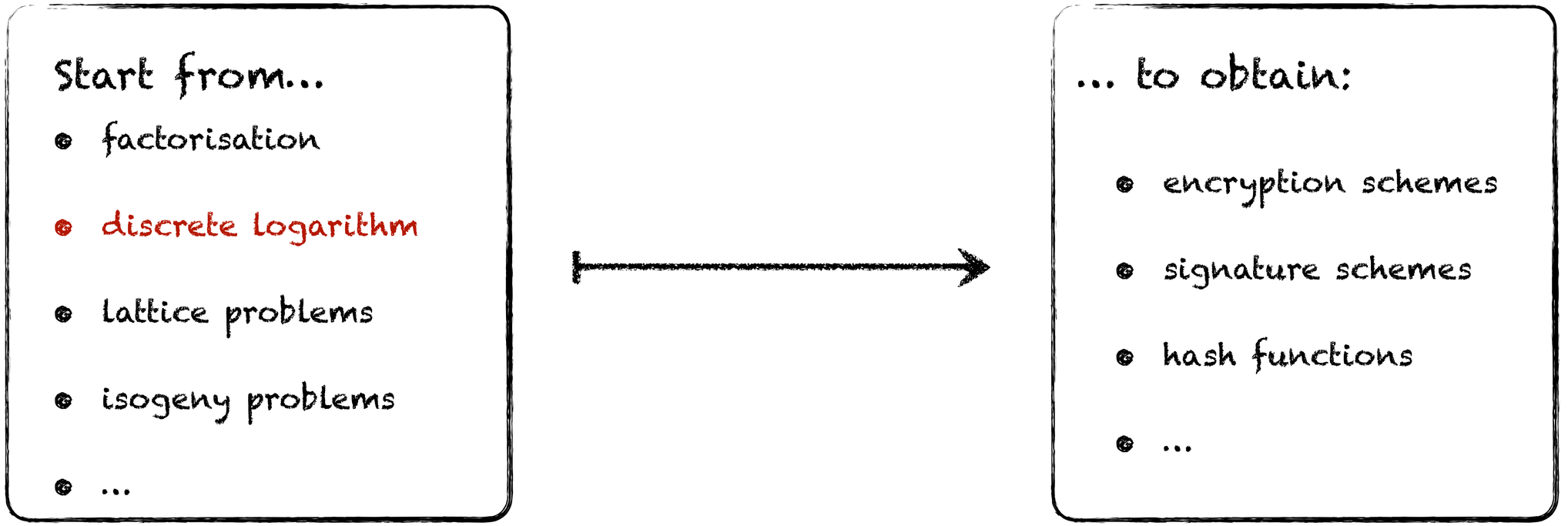
Hard problems for Cryptography

Use (hopefully) intractable problems to construct cryptographic primitives.



Hard problems for Cryptography

Use (hopefully) intractable problems to construct cryptographic primitives.



How can one attack a protocol based on the hardness of computing DL?

An example: EPID protocol in Intel SGX

- What is EPID? a protocol to allow remote attestation of a hardware platform without compromising the device's identity.

- The protocol includes a signing algorithm that uses pairings.

  - secret key includes the element $f \in_R \mathbb{Z}_q$

- How can we recover f?

  - During the protocol, consider a random secret nonce $r \in \mathbb{Z}_q$

  - Compute an exponentiation X^r

  - Outputs the element $s \leftarrow r + cf$ ($c =$ hash of known values)

How can we recover the secret f ?

Since $s \leftarrow r + cf$, if we recover r, we directly get f.

The protocol uses a 256-bit elliptic curve Fp256BN (embedding degree 12).

If we have as target X^r :

1. Solve DLP to find exponent r in 3072-bit finite field $\mathbb{F}_{p^{12}}$.

2. Look at implementation vulnerabilities during the computation of X^r.

How can we recover the secret f in EPID ?

Since $s \leftarrow r + cf$, if we recover r, we directly get f.

The protocol uses a 256-bit elliptic curve Fp256BN (embedding degree 12).

If we have as target X^r :

1. ~~Solve DLP to find exponent r in 3072-bit finite field $\mathbb{F}_{p^{12}}$.~~

2. Look at implementation vulnerabilities during the computation of X^r.

Recovering partial information on r is enough to obtain f.

Implementation vulnerabilities

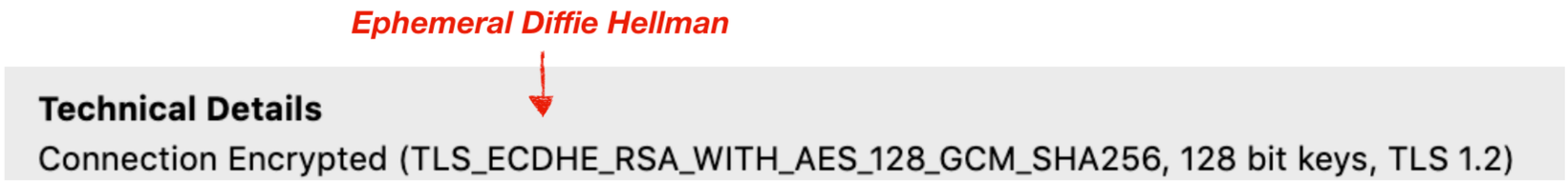
Exploiting leakage from side-channels

Vulnerable operation: modular exponentiation

Many protocols use modular exponentiation where the exponent is a secret.

Example 1: Diffie-Hellman key exchange [DH76]

- Public data: $g, g^a, g^b \in G$
- Shared key: $g^{ab} \in G$



Example 2: RSA signatures

- The victim computes $s = m^d \pmod N$ where d is a the secret exponent.

Example 3: DSA signatures

- Public data: g and p
- The victim computes a per-signature secret value k and $r = g^k \pmod p$.

Modular exponentiation

Setup: Given a finite cyclic group G of order n, a generator $g \in G$ and some element $h \in G$. For $x \in [0,n)$:

Computing modular exponentiation is easy:

$$g^x = \underbrace{g \cdot g \cdot \cdots \cdot g}_{x}$$

algorithms in $O(\log(x))$

The inverse, solving DLP can be hard (depending on the group G):

$$h = \underbrace{g \cdot g \cdot \cdots \cdot g}_{??}$$

Square and multiple algorithms

Algorithm 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: $\quad c \leftarrow c^2 \pmod{n}$ ▷ Square-
4: $\quad$ **if** $k_i = 1$ **then**
5: $\quad\quad c \leftarrow c \times x$ ▷ and-Multiply
6: **return** c

- Faster than repeated multiplications.
- Time of execution depends on the number of 1s.
- Reduce the Hamming weight of the scalar k: use wNAF representation.

Non-adjacent form (NAF) and windowed-NAF

Definition: 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 $j \geq 0$.

- Impossible to have two consecutive non-zero digits.
- Signed digits -1, 0, 1.

Windowed-NAF:

- Signed digits in a larger window: $\in [-2^w + 1, 2^w - 1]$ for a window size w.

A quick example

Example: 3 representations of 23:

- Binary: $23 = 2^4 + 2^2 + 2^1 + 2^0 = (1, 0, 1, 1, 1)$
- NAF: $23 = 2^5 - 2^3 - 2^0 = (1, 0, -1, 0, 0, -1)$
- wNAF (for $w = 3$): $23 = 2^4 + 7 \times 2^0 = (1, 0, 0, 0, 7)$

Often, implementations of fast modular exponentiation leak (partial) secret information!

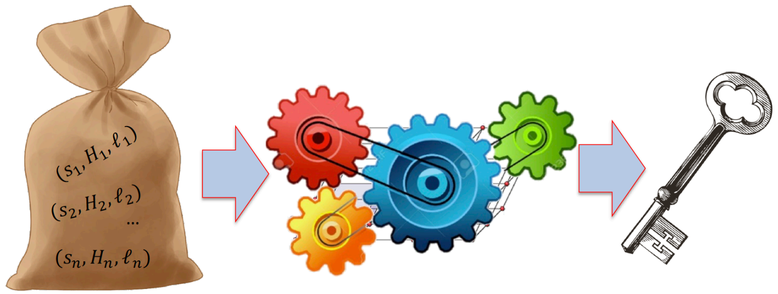
What is partial information and where does it come from?

1. Side-channel attacks, in particular cache attacks.

2. Many microarchitectural side-channel attacks use variations in execution time as the source of leakage.

3. All the public-key algorithms we discuss involve fast modular exponentiation algorithm.

In this talk, we focus on what to do with the leaked information.



Microarchitectural side-channel attacks

Goal: recover secret information by artificially creating observable contentions between CPU execution units.

- Microarchitectural elements shared between the victim and the attacker's application.
- The elements must have data-dependent state.
- The data-dependent state must be observable via side channels.

Common examples:

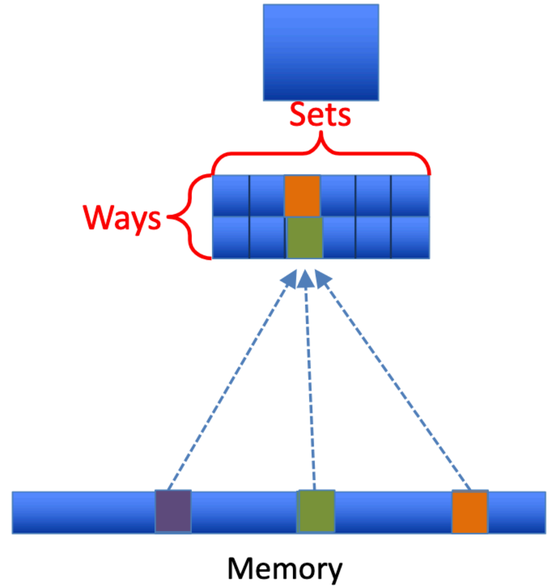
- Microarchitectural element: cache components.
- Cache attacks exploit contention on a shared cache.
- Source of leakage: execution time of certain operations (modular exponentiation).

Main idea: the attacker measures the amount of time that it takes to load information from locations in the cache.

Prime + Probe [Per05, OST06]:

1. The attacker fills cache sets by sequentially loading memory addresses that map to the same set.

2. The attacker waits for the victim to perform secret-dependent stores to the memory. When the victim stores data, it evicts some of the attacker's cache lines.



3.The attacker reloads previously cached memory addresses and measures access times to each cache sets.

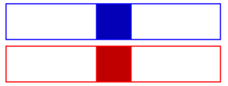
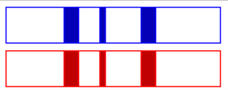
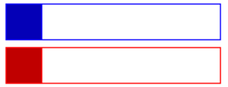
- Longer time: victim accessed the set.

Key recovery methods

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?

Methods depend on:

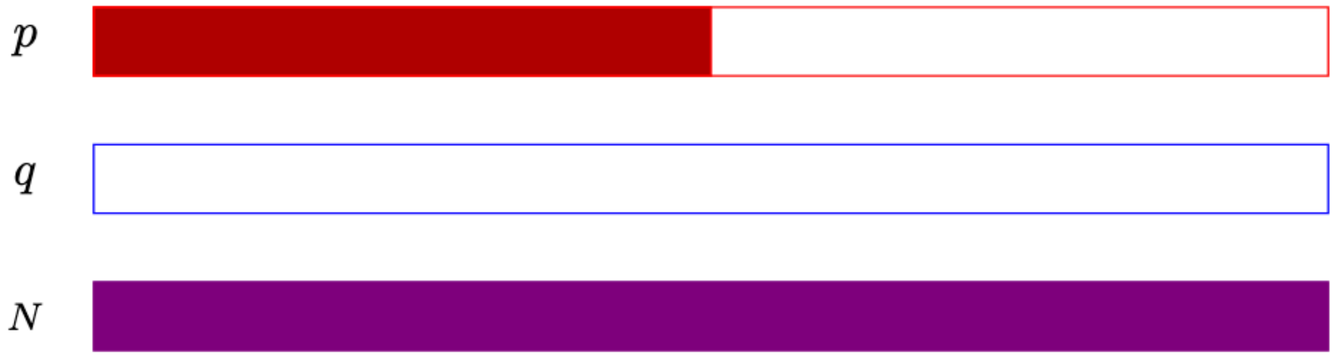
- algorithm considered,
- nature of the information leaked.

Scheme	Secret information	Bits known	Technique
RSA	$p \geq 50\%$ most significant bits		Coppersmith's method
RSA	$p \geq 50\%$ least significant bits		Coppersmith's method
RSA	p middle bits		Multivariate Coppersmith
RSA	p multiple chunks of bits		Multivariate Coppersmith
RSA	$> \log \log N$ chunks of p		Open problem
RSA	$d \pmod{p-1}$ MSBs		Coppersmith's method
RSA	$d \pmod{p-1}$ LSBs		Coppersmith's method
RSA	$d \pmod{p-1}$ middle bits		Multivariate Coppersmith
RSA	$d \pmod{p-1}$ chunks of bits		Multivariate Coppersmith
RSA	d most significant bits		Not possible
RSA	$d \geq 25\%$ least significant bits		Coppersmith's method
RSA	$\geq 50\%$ random bits of p and q		Branch and prune
RSA	$\geq 50\%$ of bits of $d \pmod{p-1}$ and $d \pmod{q-1}$		Branch and prune
(EC)DSA	MSBs of signature nonces		Hidden Number Problem
(EC)DSA	LSBs of signature nonces		Hidden Number Problem
(EC)DSA	Middle bits of signature nonces		Hidden Number Problem
(EC)DSA	Chunks of bits of signature nonces		Extended HNP
EC(DSA)	Many bits of nonce		Scales poorly
Diffie-Hellman	Most significant bits of shared secret g^{ab}		Hidden Number Problem
Diffie-Hellman	Secret exponent a		Pollard kangaroo method
Diffie-Hellman	Chunks of bits of secret exponent		Open problem

Key recovery for RSA: Two different scenarios for factorization

1. Consecutive bits known:

Coppersmith-style method



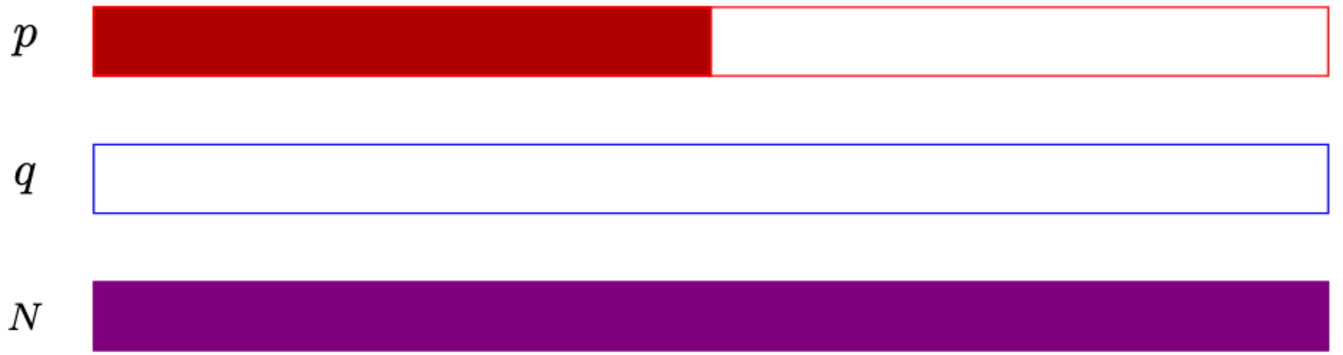
2. Random known bits:

Branch and prune method

Two different scenarios

1. Consecutive bits known:

Coppersmith-style method



2. Random known bits:

Branch and prune method

Coppersmith-style: using lattice reduction [Cop96]

Problem setup: Known RSA modulus $N = pq$.

We know a large contiguous portion of the MSBs of p.

$p =$ a $+$ r

Finding roots of a polynomial: $f(x) = x +$ a with $f(r) = p \equiv 0 \pmod{p}$

We also know r is small: $|r| < R$, and R is known.

Constructing a lattice basis:

$$B = \begin{pmatrix} R^2 & Ra & 0 \\ 0 & R & a \\ 0 & 0 & N \end{pmatrix}$$

All the quantities are known!

What happens next?

$$B = \begin{pmatrix} R^2 & Ra & 0 \\ 0 & R & a \\ 0 & 0 & N \end{pmatrix}$$

LLL →

$$v = (v_2 R^2, v_1 R, v_0)$$

(shortest vector in reduced basis)

Constructing a new polynomial: $g(x) = v_2 x^2 + v_1 x + v_0$

Find the roots of g over the integers: reconstruct $p = r + a$ and verify that $\gcd(r + a, N)$ factors N.

Why does this work?

$$B = \begin{pmatrix} R^2 & Ra & 0 \\ 0 & R & a \\ 0 & 0 & N \end{pmatrix} \begin{matrix} x(x+a) \\ x+a \\ N \end{matrix}$$

Each of these polynomials evaluated at x=r is 0 $\pmod p$

Idea: find a vector of length smaller than p.

Coppersmith's method outline

Input: $f(x) \in \mathbb{Z}[x], p \in \mathbb{Z}$ **Output:** r such that $f(r) \equiv 0 \pmod{p}$

Intermediate output: a new polynomial g such that $g(r) = 0$ over $\mathbb{Z}$.

1. $g(x) \in < f(x), p >$ so $g(r) \equiv 0 \pmod{p}$ by construction.

2. If $|r| < R$, then we can bound:

$$|g(r)| = |v_2 r^2 + v_1 r + v_0| \leq |v_2| R^2 + |v_1| R + |v_0| = ||v||_1$$

3. If $|g(r)| < p$ and $g(r) \equiv 0 \pmod{p}$ then $g(r) = 0$ over $\mathbb{Z}$.

Construct g from short vector in the lattice!

$$(\det B)^{1/\dim L} < p$$

Coppersmith's method:

1. Construct a matrix of coefficient vectors of elements of $< f(x), N >$.

2. Run a lattice basis reduction algorithm on the matrix.

3. Construct a new polynomial from the shortest vector output

4. Factor the new polynomial to find its roots.

Extending this method and limits

Known MSB or LSB or middle bits: this method works up to $R < p^{1/2}$ by increasing the lattice dimension. [Cop96]

Multiple chunks of unknown bits: $70\ \%$ of the bits of p divided into at most $\log \log N$ blocks. [HM08]

Open problem: recover an RSA modulus in sub-exponential time with more than $\log \log N$ unknown chunks.

Open problem: Recover the key from the most significant half bits of d for small exponent e.

Real-world example: Taiwan smart card IDs

Factoring RSA keys from certified smart cards: Coppersmith in the wild, Daniel J. Bernstein and Yun-An Chang and Chen-Mou Cheng and Li-Ping Chou and Nadia Heninger and Tanja Lange and Nicko van Someren

Recover 1024-bit RSA keys generated by a faulty random number generator that generated primes with predictable sequences of bits.

Key recovery for DSA and ECDSA

(EC)DSA is the elliptic curve variant of the DSA signature algorithm.

Public parameters:

- An elliptic curve E over a prime field
- A generator G of prime order n on E
- A hash function H to $\mathbb{Z}_q$

Secret key:

- An integer $\alpha \in [1, n-1]$

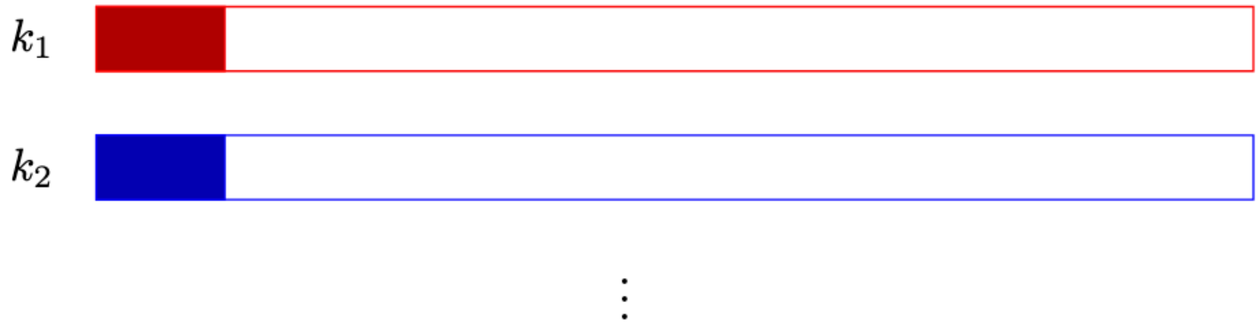
Public key:

- $p_k = [\alpha]G$: scalar multiplication of G by α

To sign a message m:

- Step 1: Randomly select a nonce $k \leftarrow_R \mathbb{Z}_n$.
- Step 2: Compute the point $(r, y) = [k]G$.
- Step 3: Compute $s = k^{-1}(H(m) + \alpha r) \pmod{n}$.
- Step 4: Output the signature (r, s).

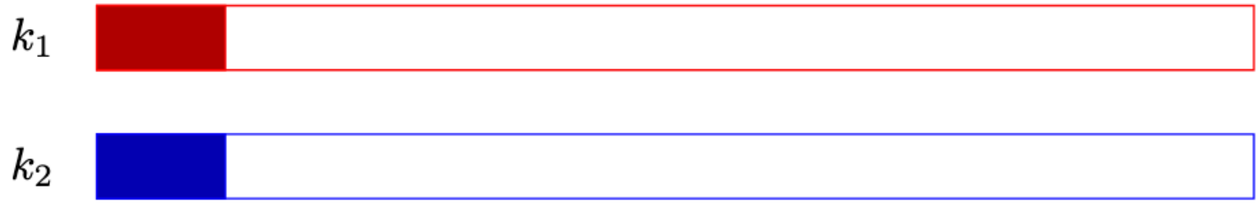
Scalar multiplication that can leak bits of k



The Hidden Number Problem [BV96]

An attacker can sign many messages and obtain a tuples $((r_i, s_i), k_i)_i$.

For each k_i we know the MSB:



$\vdots$

Obtaining a system of linear equations: $s_i \equiv k_i^{-1}(H(m_i) + \alpha r_i) \pmod{n}$

Eliminate terms and rearranging (2 signatures): $k_1 + tk_2 + u \equiv 0 \pmod{n}$

with $t = -s_1^{-1}s_2r_1r_2^{-1}$ and $u = s_1^{-1}r_1 2r_2^{-1} - s_1^{-1}h_1$

Idea: solve a linear equation with small unknowns.

Idea: solve a linear equation with small unknowns.

We have $k_1 + tk_2 + u \equiv 0 \pmod{n}$ where $|k_1|, |k_2| < K$.

Construct a Coppersmith-type lattice where a short vector will contain the unknown k_1, k_2.

$$B = \begin{pmatrix} n & 0 & 0 \\ t & 1 & 0 \\ u & 0 & K \end{pmatrix}$$

Why does this work? If the vector $v = (k_1, k_2, K)$ has norm $||v||_2 \leq (\det B)^{1/\dim L}$, it will appear in a reduced basis.

A practical example: Attacking EPID signature protocol in Intel's SGX

- What is EPID? a protocol to allow remote attestation of a hardware platform without compromising the device's identity.

- The protocol includes a signing algorithm that uses pairings.

  - secret key includes the element $f \in_R \mathbb{Z}_q$

- How can we recover f?

  - During the protocol, consider a random secret nonce $r \in \mathbb{Z}_q$

  - Compute an exponentiation X^r

  - Outputs the element $s \leftarrow r + cf$ ($c =$ hash of known values)

- Since $s \leftarrow r + cf$, if we recover r, we directly get f.

How can we recover the secret f ?

Goal: get information about the MSBs of the nonce r.

Idea: use Prime+Probe attack to count the number of iterations in the main loop of the scalar multiplication algorithm.

MultPoint(point P, window size w , scalar $r_f = \mathrm{r}$):
Initialize $P : P_0 \leftarrow O$
For $i \leftarrow 1$ to 2^{w-1} do:
    $P_i \leftarrow P \cdot P_{i-1}$
$i \leftarrow \max(j : r_j \neq 0)$ ← Start with MSB ≠ 0
$s \leftarrow P_{r_i}$
$i \leftarrow i - 1$
While $i \geq 0$ do:
    $s \leftarrow r^{2^w}$ ← w squaring operations
    $s \leftarrow s \cdot P_{r_i}$ ← Multiplication with precomputed value P_{r_i} (selected in constant-time)
    $i \leftarrow i - 1$
End while
Output: s

Main loop

Example:

- Say the nonce r is of length 256 bits.
- The nonce is recoded to a scalar of 52 digits.
- The *while loop* will have 51 iterations.

Main loop

While $i \geq 0$ do:

$s \leftarrow r^{2^w}$ — w squaring operations

$s \leftarrow s \cdot P_{r_i}$ — Multiplication with precomputed value P_{r_i} (selected in constant-time)

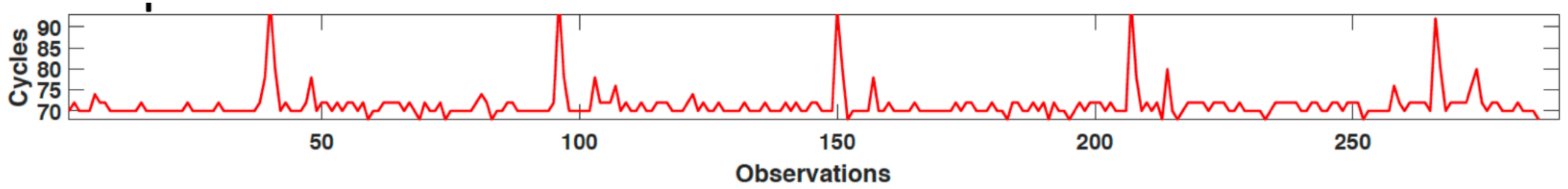
$i \leftarrow i - 1$

End while

Output: s

The attack:

- Monitor cache access patterns during the computation of the main loop.
- One period corresponds to one loop iteration.



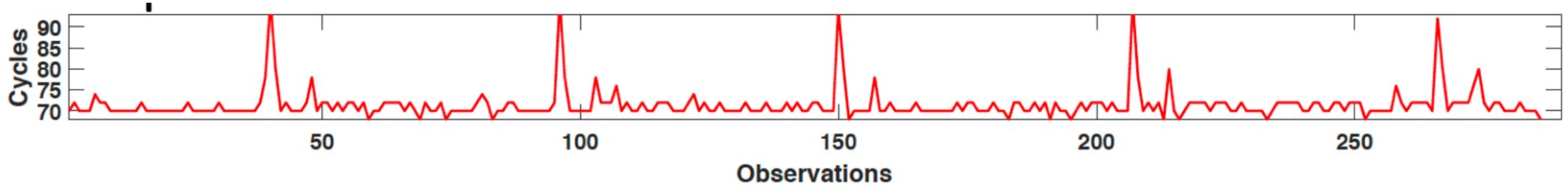
Example:

- Say the nonce r ~~is of length 256 bits.~~ has bits 256 and 255 equal to 0.
- The nonce is recoded to a scalar of ~~52 digits.~~ 51 digits.
- The *while loop* will have ~~51 loop iterations.~~ 50 loop iterations.

$$i \leftarrow \max(j : r_j \neq 0)$$ ← Start with MSB ≠ 0

The attack:

- Monitor cache access patterns during the computation of the main loop.
- One period corresponds to one loop iteration.



Analyzing the data

For many signatures and corresponding nonce r_i we know how many MSBs are equal to 0.

$|r_i| < K$ Create an instance of th Hidden Number Problem to recover the secret.

Result: using only 49-loop samples in the lattice, i.e., learning 7 MSB of the nonce, we need 38 samples to achieve above 50% success rate.

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

Attacking EPID signing algorithm

38 signatures with HNP to recover the key in 4.5 seconds

CVE-2018-3691 Detail

Current Description

Some implementations in Intel Integrated Performance Primitives Cryptography Library before version 2018 U3.1 do not properly ensure constant execution time.

– Hide Analysis Description

Analysis Description

Some implementations in Intel Integrated Performance Primitives Cryptography Library before version 2018 U3.1 do not properly ensure constant execution time.

Severity

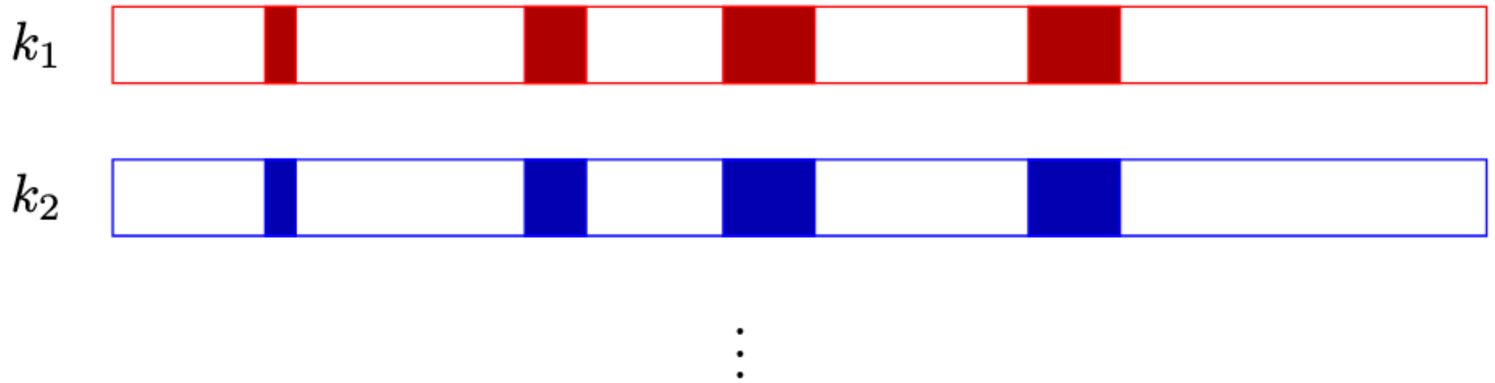
CVSS Version 3.x | CVSS Version 2.0

CVSS 3.x Severity and Metrics:

NVD **NIST:** NVD **Base Score:** 4.7 MEDIUM **Vector:** CVSS:3.0/AV:L/AC:H/PR:L/UI:N/S:U/C:H/I:N/A:N

The Extended Hidden Number Problem [HR07]

Another type of leakage…



Same idea: every unknown chunk of the nonce introduces a new variable.

Problem: much bigger lattice dimension. If m is the number of signatures, h the number of unknown chunks.

HNP: m + 2 — 37 signatures with HNP to recover the key in 4.5 seconds

EHNP: mh + 1 — 3 signatures with EHNP to recover the key in 5 days

Related publications

- *Recovering cryptographic keys from partial information, by example,* with Nadia Heninger, **Eprint 2020/1507**

- *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, at **CHES 2018**

- *A Tale of Three Signatures: practical attack of ECDSA with wNAF,* with Cécile Pierrot and Rémi Piau, at **Africacrypt 2020**

Thank you for your attention!