



Norwegian University of
Science and Technology

SIDE-CHANNEL ATTACKS 3: PUBLIC KEY CRYPTO

TTM4205 – Lecture 12

Tjerand Silde

25.09.2026

Contents

Introduction

SCA on RSA

Constant-Time Arithmetic

SCA on ECC

Testing and Certification

Summary

Contents

Introduction

SCA on RSA

Constant-Time Arithmetic

SCA on ECC

Testing and Certification

Summary

Reference Material

These slides are based on:

- ▶ The referred papers in the slides
- ▶ JPA: parts of chapters 10 and 12
- ▶ DW: parts of chapters 5 to 7 and 13
- ▶ HHH: parts of chapters 8, 11, and 14

Where This Fits

- ▶ **Lecture 9** said *what* leaks and named the attack families
- ▶ **Lecture 10** set up the hardware
- ▶ **Lecture 11** attacked AES, where the secret picks a *table index*
- ▶ **Today** we attack PKC, where the secret picks the *sequence of operations*

Black Box Crypto

We design the security of a cryptographic scheme to follow Kerckhoffs's principle: if everything about the scheme, except for the key, is known, then the scheme should be secure.

We analyze the scheme mathematically as black-box algorithms that take some (public or secret) input and give some (public or secret) output, and prove it secure concerning the algorithm description and the public data.

However, security depends on your model. In practice, it matters how these algorithms are implemented and what kind of information the *physical* system leaks about the inner workings of the algorithm computing on secret data.



Leakage

- ▶ The time it takes to compute...
- ▶ The power usage while computing...
- ▶ The electromagnetic radiation...
- ▶ The temperature variation...
- ▶ The memory pattern accessed...
- ▶ The sounds your laptop makes...
- ▶ The clock frequency the processor picks...

Attack Categories

- ▶ Remote vs physical attacks
- ▶ Software vs hardware attacks
- ▶ Passive vs active attacks
- ▶ Invasive vs non-invasive attacks

Symmetric SCA

- ▶ How AES works
- ▶ Power analysis on AES
- ▶ Correlation analysis
- ▶ Timing attacks
- ▶ Masking
- ▶ Bitslicing

What Changes for Public Key Crypto

In AES the secret selects a *table index*: the operations are always the same, and what leaks is which address is touched. Alternatively, we measure the Hamming weight of intermediate values, which correlates with the secret.

In RSA and ECC the secret selects the *operations themselves*: the exponent or scalar decides, bit by bit, whether a multiplication or an addition happens.

That is worse in one way and better in another. Worse, because a *single* trace can spell out the key. Better, because we know exactly what to fix: make the operation sequence independent of the secret (constant time code).

Contents

Introduction

SCA on RSA

Constant-Time Arithmetic

SCA on ECC

Testing and Certification

Summary

RSA Exponentiation

In the RSA cryptosystem (encryption, decryption, signing and verification), we need to compute a modular exponentiation.

If the exponent is a secret (decryption or signing) key, we must protect this value against side-channel attacks.

$$\text{Enc: } c \equiv m^e \bmod n$$

$$\text{Dec: } m \equiv c^d \equiv (m^e)^d \bmod n$$

$$\text{Sign: } s \equiv H(m)^d \bmod n$$

$$\text{Vrfy: } s^e \stackrel{?}{\equiv} H(m) \bmod n$$

The two lines with d in the exponent are the ones we have to protect.

Assumptions

In this example we assume a few things:

- ▶ the RSA primes are generated securely
- ▶ the private exponent satisfies $d \equiv e^{-1} \bmod \text{lcm}(p-1, q-1)$
- ▶ we have a way of representing large integers

Weaknesses and Defenses

In the following slides we will look at the common ways to compute modular exponentiation. For each algorithm, try to come up with attacks and defenses for the algorithm.

Square and Multiply

```
1  # compute m = c**d mod n
2  def squareAndMultiply(c, d, n):
3      m = 1
4
5      for i in range(len(d)):
6          m = m * m % n
7
8          if ( d[i] == 1 ):
9              m = m * c % n
10
11     return m
```

What the Trace Shows

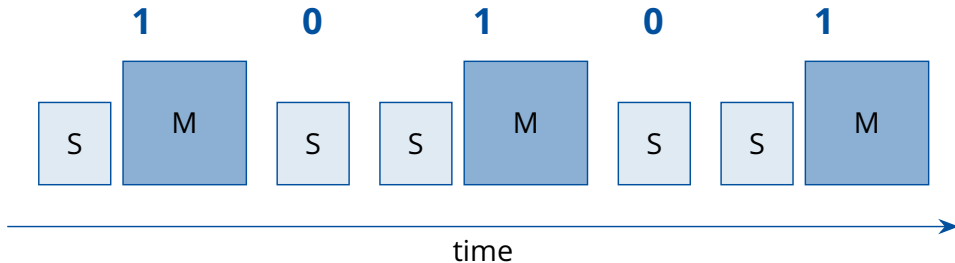


Figure: The branch `if d[i] == 1` is visible in the trace: a squaring followed by a multiplication means a one bit, a lone squaring means a zero. We only need one trace, and then the whole exponent leaks directly.

Potential Weaknesses

The following might trivially leak the key:

- ▶ timing or power traces might leak the 1's in d
- ▶ multiplication might not be constant time
- ▶ modular reduction might not be constant time

Potential Defenses

We must at least ensure the following:

- ▶ the algorithm must be independent of the 1's in d
- ▶ big integer multiplication must be constant time
- ▶ modular reduction must be constant time

Assume for now that the two latter operations are constant time.

Square and Always Multiply

```
1  # compute m = c**d mod n
2  def squareAndAlwaysMultiply(c, d, n):
3      m, x = 1, 1
4
5      for i in range(len(d)):
6          m = m * m % n
7
8          if ( d[i] == 1 ):
9              m = m * c % n
10         else:
11             x = m * c % n # dummy: result thrown away
12
13     return m
```

Potential Weaknesses

- ▶ dummy operations might leak memory information
- ▶ “smart” compilers might skip dummy operations
- ▶ fault injections might expose dummy operations

The fault attack is important: glitch one multiplication. If the output is still correct, that multiplication was the dummy, so the bit was 0. Repeat once per multiplication, one glitch per execution, and every bit of d falls out.

Potential Defenses

- ▶ make the result dependent on every operation
- ▶ perform the same operations independent of d

Montgomery Ladder

```
1  # compute m = c**d mod n
2  def MontgomeryLadder(c, d, n):
3      m1, m2 = 1, c
4
5      for i in range(len(d)):
6
7          if ( d[i] == 1 ):
8              m1 = m1 * m2 % n
9              m2 = m2 * m2 % n
10
11         else:
12             m2 = m1 * m2 % n
13             m1 = m1 * m1 % n
14
15     return m1
```

Why the Ladder Is Regular

Every iteration does exactly one squaring and one multiplication, in the same order, whatever the bit is. Only *which variable* receives the result changes.

The loop maintains the invariant

$$m_2 \equiv m_1 \cdot c \pmod{n},$$

so the correct values are being built, and no dummy result is ever discarded.

Every operation contributes to the output: a fault in any of them corrupts the result, which is exactly what square-and-always-multiply failed to achieve.

Potential Weaknesses

There might still be issues:

- ▶ if c is chosen adaptively, many power traces might leak d

Potential Defenses

Randomization to the rescue:

- ▶ randomize the computation to make it independent of c

Randomized Montgomery Ladder

```
1  # compute m = c**d mod n, where e*d = 1 mod phi
2  def randMontgomeryLadder(c, e, d, phi, n):
3      r1 = secrets.randbelow(n) # fresh blinding factor
4      r2 = squareAndMultiply(r1, e, n) # r1**e mod n
5      r1Inv = MontgomeryLadder(r1, phi-1, n) # r1**(-1) mod n
6
7      m1, m2 = 1, c * r2 % n # ladder start, blinded base
8
9      for i in range(len(d)):
10         if ( d[i] == 1 ):
11             m1, m2 = m1 * m2 % n, m2 * m2 % n
12         else:
13             m2, m1 = m1 * m2 % n, m1 * m1 % n
14
15     return m1 * r1Inv % n # unblind
```

Potential Weaknesses

There might still be issues:

- ▶ if the key is fixed, many power traces might leak d

Potential Defenses

Randomization to the rescue (again):

- ▶ randomize the exponent to mask the key d

Doubly Randomized Montgomery Ladder

```
1  # compute m = c**d mod n, where e*d = 1 mod phi
2  def randRandMontgomeryLadder(c, e, d, phi, n):
3      r1 = secrets.randbelow(n)
4      r2 = squareAndMultiply(r1, e, n)
5      r1Inv = MontgomeryLadder(r1, phi-1, n)
6
7      r = secrets.randbelow(n)
8      dNew = bits(d + r * phi) # same exponent mod phi, new bits
9      m1, m2 = 1, c * r2 % n # ladder start, blinded base
10     for i in range(len(dNew)):
11         if ( dNew[i] == 1 ):
12             m1, m2 = m1 * m2 % n, m2 * m2 % n
13         else:
14             m2, m1 = m1 * m2 % n, m1 * m1 % n
15     return m1 * r1Inv % n
```

Two Randomizations, Two Jobs

Base blinding ($c \mapsto c \cdot r^e$) makes every trace operate on a value the adversary does not know, so traces cannot be averaged against a chosen ciphertext.

Exponent blinding ($d \mapsto d + r\varphi$) changes the *bit pattern* that the loop follows, so traces of the same key no longer line up with each other.

Neither is sufficient alone, and both cost: the blinding factors have to be produced, and $d + r\varphi$ is longer than d , so the loop runs for more iterations.

Summary

Protecting secret key computations is difficult. We need:

- ▶ all binary operations to be constant time
- ▶ the algorithmic operations to be constant time
- ▶ correctness of output to depend on all operations
- ▶ the base element to be randomized (masked)
- ▶ the exponent to be randomized (masked)

Contents

Introduction

SCA on RSA

Constant-Time Arithmetic

SCA on ECC

Testing and Certification

Summary

Representing Large Integers

This is usually done by representing them as a list of machine words of 32 or 64 bits. Arithmetic is then done word by word, carrying whenever a word overflows.

For example, an RSA-4096 modulus can be represented as a list of 128 words of 32 bits, or 64 words of 64 bits.



Why CT Arithmetic Matters

Execution time or power consumption must not depend on secret data.

- ▶ Multi-precision addition and subtraction can take longer if carries propagate through many words.
- ▶ Naive modular reduction uses `if (x >= p)` branches.
- ▶ Division or inversion loops have variable length.

Goal: ensure identical instruction sequence and timing for all inputs.

A single MUL instruction multiplies two 32-bit words and writes the full 64-bit product as two 32-bit halves, the high half and the low half. On x86 this is constant time.

Disclaimer 1: this depends on the machine you are using.

Disclaimer 2: this depends on the compiler you are using.



Figure: <https://www.bearssl.org/constanttime.html>

Not Every Multiplier Is Constant Time

CPU	32 → 32	32 → 64	MUL31	64 → 64
ARM7T, ARM9T (A32)	N	N	Y	S−
ARM7T, ARM9T (Thumb)	N	S−	S−	S−
ARM9E, ARM10E, ARM11	Y	Y	Y	S+
Cortex-A, Cortex-R (A32)	Y	Y	Y	S+
Cortex-M0/M0+/M1	Y	S−	S−	S−
Cortex-M3	Y	N	N	S−
Cortex-M4	Y	Y	Y	S+

Y = constant time, N = *not* constant time, S+ / S− = no hardware opcode, so the compiler supplies a software routine that is probably / probably *not* constant time. See <https://www.bearssl.org/ctmul.html>.

Contents

Introduction

SCA on RSA

Constant-Time Arithmetic

SCA on ECC

Testing and Certification

Summary

Recall: Elliptic Curve Diffie-Hellman

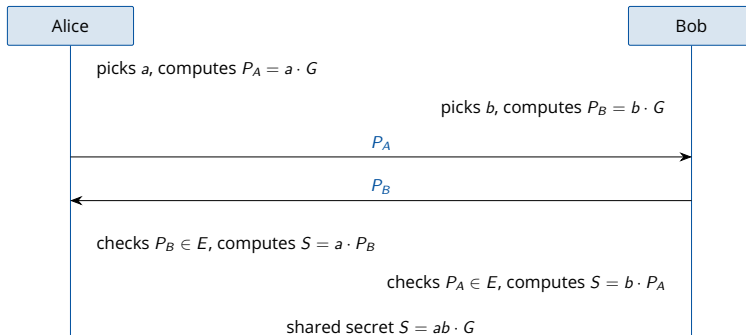


Figure: Public: the curve E and the base point G . Secret: the scalars a and b . The only secret-dependent computation is the scalar multiplication.

Side-channel attacks on ECC often use the same principles as RSA:

- ▶ The secret influences which operations are executed
- ▶ Timing or power traces can reveal those secrets

Question: If we replace modular exponentiation $m^d \bmod n$ with scalar multiplication kP , what might change in terms of leakage?

SCA on ECC: Key Differences from RSA

- ▶ Formulas depend on the curve parameters, so arithmetic rules differ per curve.
- ▶ Addition and doubling use *different* formulas, and in affine coordinates both need an inversion.
- ▶ Operations depend on the input point, so the sequence of field operations can change.
- ▶ Special cases ($P = \mathcal{O}$, $P = -Q$, $P = Q$) are handled by extra branches.

Implication: Unlike modular exponentiation, scalar multiplication may leak through branching or timing tied to point-dependent formulas.

Naive Scalar Multiplication

```
1  # compute R = k*P on the curve E
2  def pointMultiplication(P, k, E):
3      R = 0 # the point at infinity
4      Q = P
5
6      for i in range(len(k)): # bits of k, least significant first
7          if ( k[i] == 1 ):
8              R = pointAddition(R, Q, E)
9
10         Q = pointDoubling(Q, E)
11
12     return R
```

The same shape as square-and-multiply, and the same leak.

Potential Weaknesses

The following might leak the key:

- ▶ Double-and-add performs an addition only when a bit of k equals 1, so power or timing traces directly reveal this pattern.
- ▶ Multiplications and reductions may take longer for certain operands, leaking through timing.
- ▶ Point addition and doubling often require inversions that depend on the inputs.

Potential Defenses

We must at least ensure the following:

- ▶ Use algorithms such as the Montgomery ladder so that every loop performs both a doubling and an addition, regardless of the key bit.
- ▶ Ensure modular multiplication and reduction routines take identical time for all operands.
- ▶ Switch to projective coordinates so that expensive inversions are replaced by constant-time multiplications.
- ▶ Add blinding of scalars or points and verify that input points are valid on the curve.

Why Projective Coordinates Help

In affine coordinates, adding two points needs a slope, and a slope needs a modular inversion. Inversion is the slowest field operation, and the classical algorithms for it are *not* constant time.

Writing a point as $(X : Y : Z)$ with $x = X/Z$, $y = Y/Z$ postpones the division: the whole scalar multiplication runs with multiplications and squarings only, and a single inversion converts back at the end.

As a bonus, the representation is *redundant*: $(X : Y : Z)$ and $(\lambda X : \lambda Y : \lambda Z)$ are the same point. That freedom is what we randomize below.



Blinded Scalar Multiplication

Additive splitting. Alice draws r_A at random and computes $P_A = (a + r_A) \cdot G - r_A \cdot G$, and likewise $S_A = (a + r_A) \cdot P_B - r_A \cdot P_B$. Neither scalar multiplication ever uses a itself.

Scalar randomization. With n the order of G , replace a by $a' = a + r \cdot n$ for random r , which leaves $a' \cdot G = a \cdot G$ but changes the bit pattern in the loop.

The first doubles the work; the second lengthens the loop by the size of r . Both destroy the alignment that averaging attacks depend on.

Randomized Point Representation

Point blinding. Alice draws a random curve point R_A and computes $S_A = a \cdot (P_B + R_A) - a \cdot R_A$, so the ladder never processes a publicly known point.

Projective randomization. Represent the input as $(X : Y : Z) = (\lambda x : \lambda y : \lambda)$ for a random $\lambda \in \mathbb{F}_p^*$, run the ladder, and recover $(X/Z, Y/Z)$ at the end.

Every intermediate coordinate is then multiplied by an unknown factor, so the Hamming weights an attacker measures are randomized at essentially no cost.

Doubly Randomized Scalar Multiplication

```
1  def doublyRandomizedMul(P, k, E, n, p):
2      r = secrets.randbelow(n)
3      kPrime = k + r * n # same result, new bit pattern
4
5      lam = secrets.randbelow(p-1) + 1
6      x, y = P
7      Pr = (lam * x % p, lam * y % p, lam) # projective (X:Y:Z)
8      R0, R1 = O, Pr # ladder start: (infinity, P)
9      for b in bits(kPrime): # Montgomery ladder, most significant first
10         if b == 1:
11             R0, R1 = pointAddition(R0, R1, E), pointDoubling(R1, E)
12         else:
13             R1, R0 = pointAddition(R0, R1, E), pointDoubling(R0, E)
14
15     X, Y, Z = R0 # back to affine
16     Zinv = inverse(Z, p)
17     return (X * Zinv % p, Y * Zinv % p)
```

Designing the Problem Away

X25519 and Ed25519

Clamping. The scalar has its low three bits cleared and bit 254 set. The first makes the scalar a multiple of 8, the curve's *cofactor*, so points of small order contribute nothing; the second fixes the bit length, so the ladder always runs the same number of iterations regardless of the key.

Complete formulas. On the twisted Edwards form, one addition formula is correct for *all* inputs, including $P = Q$ and the identity, so there are no special cases and therefore no branches.

A friendly prime. $p = 2^{255} - 19$ makes reduction a fixed sequence of multiplications and additions, with no data-dependent path.

Deterministic nonces in Ed25519: $r = H(\text{key prefix} \parallel m)$; there is no RNG-fail.

Two Philosophies

Patch the implementation. Take a curve chosen for other reasons, then add regular ladders, blinding, validation and constant-time field arithmetic on top. Every one of those is a chance to get it wrong.

Choose the parameters so the mistakes are unavailable. Pick a curve, a coordinate system and an encoding for which the fast implementation is already the safe one.

The second is why X25519 and Ed25519 are recommended today, and it is the same lesson as bitslicing in the previous lecture: the best countermeasure is a design where the dangerous operation does not exist.

Comparative Study of ECC Libraries for Embedded Devices

Tjerand Silde

Norwegian University of Science and Technology, Trondheim, Norway
`tjerand.silde@ntnu.no`, `www.tjerandsilde.no`

Figure: <https://tjerandsilde.no/files/Comparative-Study-of-ECC-Libraries-for-Embedded-Devices.pdf>

Contents

Introduction

SCA on RSA

Constant-Time Arithmetic

SCA on ECC

Testing and Certification

Summary

Dude, is my code constant time?

Oscar Reparaz, Josep Balasch and Ingrid Verbauwhede
KU Leuven/COSIC and imec
Leuven, Belgium

Figure: <https://eprint.iacr.org/2016/1123.pdf>



Testing, Not Trusting

`dudect` takes a black-box statistical approach: run the implementation on two input classes, one fixed and one random, and test whether the two timing distributions differ. If they do, the code is not constant time.

It needs no model of the hardware, which is exactly the point: the leakage you failed to predict is the one that gets you.

Related tools mark secrets as uninitialized memory and let a dynamic analyzer report every branch or memory index that depends on them: `ctgrind`, `MemSan`, `TIMECOP`.

None of these prove absence of leakage. They only find leaks.

FIPS 140-3

Security Requirements for Cryptographic Modules



Date Published: March 22, 2019

Supersedes: [FIPS 140-2 \(12/03/2002\)](#)

Planning Note (05/01/2019): 

See the [FIPS 140-3 Transition](#) project for the following information:

- [FIPS 140-3 Transition Schedule](#)
- Supporting [SP 800-140x documents](#) that modify requirements of ISO/IEC 19790:2012 and ISO/IEC 24759:2017

Author(s)

National Institute of Standards and Technology

Figure: <https://csrc.nist.gov/pubs/fips/140-3/final>



Certification

FIPS 140-3 sets four security levels for cryptographic modules, from “approved algorithms, no physical protection” up to tamper detection and response, environmental failure protection, and hardened enclosures.

Levels 3 and 4 require resistance to non-invasive attacks, which is where power and EM analysis enter a certification document at all.

A certificate says a laboratory tested a specific module, in a specific configuration, against a specific list. It does not say the module is secure against an attack invented afterwards.

Contents

Introduction

SCA on RSA

Constant-Time Arithmetic

SCA on ECC

Testing and Certification

Summary

What to Remember

- ▶ In PKC the secret chooses the *operations*, so one trace can be enough
- ▶ Regularity first: same operations, same order, every iteration
- ▶ Dummy operations are not regularity, and a fault tells them apart
- ▶ Blinding: randomize the base, the exponent or scalar, and the coordinates
- ▶ The leak often lives one level down, in the multiplier or the reduction
- ▶ “Constant time” is a property of the binary, not of the source

Questions?