Norwegian University of Science and Technology

SIDE-CHANNEL ATTACKS 3: PUBLIC KEY CRYPTO

TTM4205 – Lecture 14

Caroline Sandsbråten

06.10.2025

Contents

Previous Lecture

SCA on RSA

CT Arithmetic

SCA on ECC

Interesting papers

Contents

Previous Lecture

SCA on RSA

CT Arithmetic

SCA on ECC

Interesting papers

Black Box Crypto

We design the security of a cryptographic scheme to follow Kerckhoff'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...

Attack Categories

- Remote vs physical attacks
- Software and 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.

Contents

Previous Lecture

SCA on RSA

CT Arithmetic

SCA on ECC

Interesting papers

RSA Exponentiation

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

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

Dec: $m \mod n \equiv c^d \equiv (m^e)^d$

Enc: $c \equiv m^e \mod n$

Sign: $s \equiv H(m)^d \mod n$

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

Assumptions

In this example we assume a few things:

- the RSA primes are generated securely
- order phi is computed as $\mathrm{lcm}(p-1, q-1)$
- we have a way of representing larger 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 = c
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
```

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:

- algorithm must be independent of the 1's in d
- bit int multiplication must be constant time
- modular reduction must be constant time

Assume 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 = c, c
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          else:
12              x = m * c % n
13
14      return m
```

Potential Weaknesses

- dummy operations might leak memory information
- "smart" compilers might skip dummy operations
- fault injections might expose dummy operations

Potential Defenses

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

Montgomery Ladder

```python
# compute m = c**d mod n
def MontgomeryLadder(c, d, n):
    m1, m2 = c, c * c % n

    for i in range(len(d)):

        if ( d[i] == 1 ):
            m1 = m1 * m2 % n
            m2 = m2 * m2 % n

        else:
            m2 = m1 * m2 % n
            m1 = m1 * m1 % n

    return m1
```

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
2  # we have e*d = 1 mod phi
3  def randMontgomeryLadder(c, e, d, phi, n):
4
5      r1 = secrets.randbelow(n)
6      r2 = squareAndMultiply(r1, e, n)
7      r1Inv = MontgomeryLadder(r1, phi-1, n)
8
9      m1 = c * r2 % n
10     m2 = m1 * m1 % n
11
12     for i in range(len(d)):
13
14         if ( d[i] == 1 ):
15             m1 = m1 * m2 % n
16             m2 = m2 * m2 % n
17
18         else:
19             m2 = m1 * m2 % n
20             m1 = m1 * m1 % n
21
22     m1 = m1*r1Inv % n
23     return m1
```

Potential Weaknesses

There might still be issues:

- if 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
2   # we have e*d = 1 mod phi
3   def randRandMontgomeryLadder(c, e, d, phi, n, t):
4
5       r1 = secrets.randbelow(n)
6       r2 = squareAndMultiply(r1, e, n)
7       r1Inv = MontgomeryLadder(r1, phi-1, n)
8
9       r = secrets.randbelow(t)
10      # get dNew = d + r * phi
11      dNew = convert(d, r, phi)
12
13      m1 = c * r2 % n
14      m2 = m1 * m1 % n
15
16      for i in range(len(dNew)):
17
18          if ( dNew[i] == 1 ):
19              m1 = m1 * m2 % n
20              m2 = m2 * m2 % n
21
22          else:
23              m2 = m1 * m2 % n
24              m1 = m1 * m1 % n
25
26      m1 = m1*r1Inv % n
27      return m1
```

Summary

Protecting secret key computations are 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

Previous Lecture

SCA on RSA

CT Arithmetic

SCA on ECC

Interesting papers

Representing Large Integers

This is usually done by representing them as a list of integers of 32 or 64 bits. Binary operations is then done over the list of integers and must remember the carry when it overflows.

For example, a RSA-4096 moduli can be represented using a list of 128 integers of 32 bits or 64 integers 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.

Intel IMUL

Takes in two 32 bit integers to be multiplied and outputs two 32 bit integers representing the upper and lower 32 bits of the product. This operation is constant time.

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

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

Bear SSL

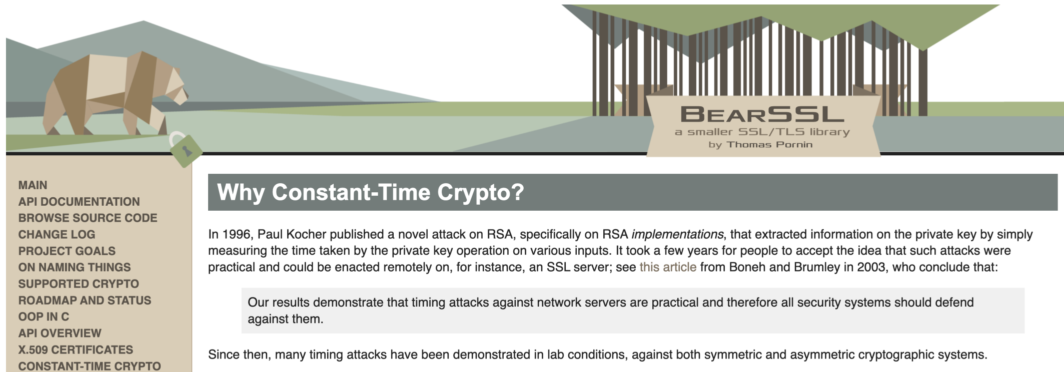


Figure: `https://www.bearssl.org/constanttime.html`

Arm MUL

CPU type	32→32	32→64	MUL31	64→64
ARM7T (A32)	N	N	Y	S-
ARM7T (Thumb)	N	S-	S-	S-
ARM9T (A32)	N	N	Y	S-
ARM9T (Thumb)	N	S-	S-	S-
ARM9E	Y	Y	Y	S+
ARM10E	Y	Y	Y	S+
ARM11	Y	Y	Y	S+
Cortex-A (A32)	Y	Y	Y	S+
Cortex-A (A64)	?	?	?	?
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+

Figure: `https://www.bearssl.org/ctmul.html`

Montgomery Modular Multiplication

```
function REDC is
    input: Integers R and N with gcd(R, N) = 1,
           Integer N′ in [0, R − 1] such that NN′ ≡ −1 mod R,
           Integer T in the range [0, RN − 1].
    output: Integer S in the range [0, N − 1] such that S ≡ TR^−1 mod N

    m ← ((T mod R)N′) mod R
    t ← (T + mN) / R
    if t ≥ N then
        return t − N
    else
        return t
    end if
end function
```

Figure: https://en.wikipedia.org/wiki/Montgomery_modular_multiplication

Montgomery Modular Multiplication

```
28 void
29 br_i31_montymul(uint32_t *d, const uint32_t *x, const uint32_t *y,
30         const uint32_t *m, uint32_t m0i)
31 {
32         size_t len, len4, u, v;
33         uint64_t dh;
34 
35         len = (m[0] + 31) >> 5;
36         len4 = len & ~(size_t)3;
37         br_i32_zero(d, m[0]);
38         dh = 0;
39         for (u = 0; u < len; u ++) {
40                 uint32_t f, xu;
41                 uint64_t r, zh;
42 
43                 xu = x[u + 1];
44                 f = MUL31_lo((d[1] + MUL31_lo(x[u + 1], y[1])), m0i);
45 
46                 r = 0;
47                 for (v = 0; v < len4; v += 4) {
48                         uint64_t z;
49 
50                         z = (uint64_t)d[v + 1] + MUL31(xu, y[v + 1])
51                                 + MUL31(f, m[v + 1]) + r;
52                         r = z >> 31;
53                         d[v + 0] = (uint32_t)z & 0x7FFFFFFF;
54                         z = (uint64_t)d[v + 2] + MUL31(xu, y[v + 2])
55                                 + MUL31(f, m[v + 2]) + r;
56                         r = z >> 31;
57                         d[v + 1] = (uint32_t)z & 0x7FFFFFFF;
58                         z = (uint64_t)d[v + 3] + MUL31(xu, y[v + 3])
59                                 + MUL31(f, m[v + 3]) + r;
60                         r = z >> 31;
61                         d[v + 2] = (uint32_t)z & 0x7FFFFFFF;
62                         z = (uint64_t)d[v + 4] + MUL31(xu, y[v + 4])
63                                 + MUL31(f, m[v + 4]) + r;
64                         r = z >> 31;
65                         d[v + 3] = (uint32_t)z & 0x7FFFFFFF;
66                 }
67                 for (; v < len; v ++) {
68                         uint64_t z;
69 
70                         z = (uint64_t)d[v + 1] + MUL31(xu, y[v + 1])
71                                 + MUL31(f, m[v + 1]) + r;
72                         r = z >> 31;
73                         d[v] = (uint32_t)z & 0x7FFFFFFF;
74                 }
75 
76                 zh = dh + r;
77                 d[len] = (uint32_t)zh & 0x7FFFFFFF;
78                 dh = zh >> 31;
79         }
80 
81         /*
82          * We must write back the bit length because it was overwritten in
83          * the loop (not overwriting it would require a test in the loop,
84          * which would yield bigger and slower code).
85          */
86         d[0] = m[0];
87 
88         /*
89          * d[] may still be greater than m[] at that point; notably, the
90          * 'dh' word may be non-zero.
91          */
92         br_i31_sub(d, m, NEQ(dh, 0) | NOT(br_i31_sub(d, m, 0)));
93 }
```

Constant Time IF

A possible way to compute an IF in constant time:

$$(t < N) \cdot t + (1 - (t < N)) \cdot (t - N)$$

Disclaimer: "smart" compilers might make it a regular IF.

Contents

Previous Lecture

SCA on RSA

CT Arithmetic

SCA on ECC

Interesting papers

EC Diffie-Hellman

Recall: ECDH, basic ECC key exchange.

1: **Inputs:** Elliptic curve E, base point G, private key a for Alice, private key b for Bob, prime p
2: **Outputs:** Shared secret S
3: **procedure** ECDH
4: Alice computes $P_A := a \cdot G$ on curve E ▷ Public key computation
5: Alice sends P_A to Bob, Bob verifies P_A is on curve E
6: Bob computes $P_B := b \cdot G$ on curve E ▷ Public key computation
7: Bob sends P_B to Alice, Alice verifies P_B is on curve E
8: Alice computes $S := a \cdot P_B$ ▷ Shared secret
9: Bob computes $S := b \cdot P_A$ ▷ Shared secret
10: **Both Alice and Bob now share the same secret** S
11: **end procedure**

SCA on ECC

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 formulas differ, and doubling often requires inversion.
- Operations depend on input points so the sequence of field operations can change.

SCA on ECC

Key differences from RSA

- Formulas depend on the curve parameters, so arithmetic rules differ per curve.
- Addition and doubling formulas differ, and doubling often requires inversion.
- Operations depend on input points so the sequence of field operations can change.

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

SCA on ECC: Countermeasures

To achieve side-channel resistance, ECC implementations must ensure that every iteration of scalar multiplication behaves identically.

- Use unified addition formulas that work for both addition and doubling.
- Use projective coordinates to remove inversions inside the main loop.
- Compute any required inversions in constant time using Fermat's trick.
- Validate input points and curve parameters to avoid malicious inputs.

Goal: All operations should be independent of the secret scalar k and the intermediate point values.

Scalar multiplication

Zoom in: Naive scalar multiplication

1: **Inputs:** Point $P = (x, y)$ on elliptic curve E, scalar k (private key), prime p
2: **Outputs:** Point $kP = (x_n, y_n)$
3: **procedure** PointMultiplication
4: $R = \mathcal{O}$ ▷ Initialize to the identity
5: $Q = P$ ▷ Initialize Q as the input point P
6: **for** each bit k_i of k from left to right **do**
7: **if** $k_i = 1$ **then**
8: $R =$ PointAddition(R, Q) ▷ Add Q to R if bit is 1
9: **end if**
10: $Q =$ PointDoubling(Q) ▷ Double the point Q each iteration
11: **end for**
12: **return** R
13: **end procedure**

Potential Weaknesses

The following might leak the key:

- The double-and-add algorithm performs additions only when a bit of k equals 1. Power or timing traces directly reveal this pattern.
- Multiplications and reductions may take longer for certain operands, leaking information 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.

And other similar defenses as for the square and multiply algorithm earlier in the lecture.

Blinded Scalar Multiplication (Alice PoV)

1: **Inputs:** Elliptic curve E, base point G, private key a for Alice, private key b for Bob, prime p
2: **Outputs:** Shared secret S
3: **procedure** ECDH
4: Alice generates a random blinding factor $r_A \in [1, p-1]$
5: Alice computes the masked public key $P_A = (a + r_A) \cdot G - r_A \cdot G$
6: Alice sends P_A to Bob, receives P_B from Bob, Alice and Bob verifies P_B and P_A is on curve E, respectively
7: Alice computes the shared secret as $S_A = (a + r_A) \cdot P_B - r_A \cdot P_B$
8: **Both Alice and Bob now share the same secret** $S = S_A = S_B$
9: **end procedure**

Randomized Point Addition (Alice PoV)

1: **Inputs:** Elliptic curve E, base point G, private key a for Alice, private key b for Bob, prime p
2: **Outputs:** Shared secret S
3: **procedure** ECDH
4: Alice generates a random point R_A on the elliptic curve
5: Alice computes the masked public key $P_A = a \cdot (G + R_A) - a \cdot R_A$
6: Alice sends P_A to Bob, receives P_B from Bob, Alice and Bob verifies P_B and P_A is on curve E, respectively
7: Alice computes the shared secret as $S_A = a \cdot (P_B + R_A) - a \cdot R_A$
8: **Both Alice and Bob now share the same secret** $S = S_A = S_B$
9: **end procedure**

Doubly Randomized Point Multiplication

1: **Inputs:** Point $P = (x, y)$ on elliptic curve $E : y^2 = x^3 + ax + b$, scalar k, prime p, curve order n
2: **Outputs:** Point $kP = (x_k, y_k)$
3: Random values $r \in \mathbb{Z}$ and $z_r \in \mathbb{R}^+$
4: **procedure** DoublyRandomizedPointMultiplication
5: **Step 1: Randomize the scalar**
6: Generate a random integer $r \in [1, n-1]$
7: Let $k' = k + r \cdot n$
8: **Step 2: Randomize the point coordinates**
9: Generate a random non-zero scalar z_r
10: Transform the input point:

$$P_r = (x_r, y_r) = (z_r \cdot x, z_r \cdot y) \mod p$$

11: **Step 3: Montgomery Ladder with Randomized Inputs**
12: Initialize $R_0 = P_r$ and $R_1 = 2P_r$
13: **for** each bit k'_i of k' from left to right **do**
14: **if** $k'_i = 1$ **then**
15: Swap R_0 and R_1
16: **end if**
17: $R_0 =$ PointDoubling(R_0)
18: $R_1 =$ PointAddition(R_0, P_r)
19: **end for**
20: **Step 4: Remove the randomization**
21: Recover the coordinates by dividing by the random scalar:

$$(x_k, y_k) = (R_0.x/z_r, R_0.y/z_r) \mod p$$

22: **return** (x_k, y_k)
23: **end procedure**

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

Previous Lecture

SCA on RSA

CT Arithmetic

SCA on ECC

Interesting papers

Optical Cryptanalysis: Recovering Cryptographic Keys from Power LED Light Fluctuations

Ben Nassi[1,2], Ofek Vayner[1], Etay Iluz[1], Dudi Nassi[1], Or Hai Cohen[1], Jan Jancar[3], Daniel Genkin[4], Eran Tromer[5], Boris Zadov[1], Yuval Elovici[1]

[1] Ben-Gurion University of the Negev, [2] Cornell Tech, [3] Masaryk University, [4] Georgia Tech, [5] Columbia University
{nassib, ofekvay, etayil, nassid, ora2, zadov}@post.bgu.ac.il, bn267@cornell.edu, elovici@bgu.ac.il, genkin@gatech.edu, 445358@mail.muni.cz, et2555@columbia.edu

Paper: https://eprint.iacr.org/2023/1068

Video-Based Cryptanalysis: Extracting Cryptographic Keys from Video Footage of a Device's Power LED

Ben Nassi[1,2], Etay Iluz[2], Or Cohen[2], Ofek Vayner[2], Dudi Nassi[2], Boris Zadov[2], Yuval Elovici[2]

[1]Cornell Tech, [2]Ben-Gurion University of the Negev
bn267@cornell.edu, {nassib, etayil, ora2, ofekvay, nassid, zadov}@post.bgu.ac.il, elovici@bgu.ac.il
Website - https://www.nassiben.com/video-based-crypta

Paper: `https://eprint.iacr.org/2023/923`

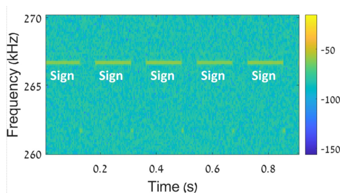
Where did the idea come from?

- Had previously used a photo diode to recover speech based on intensity of LED light.
- How did they do this?

Paper: https://iacr.org/submit/files/slides/2024/rwc/rwc2024/1/slides.pptx

Video Based Cryptanalysis

- Detecting the when an ECDSA signing operation starts and finishes.
- How did they do this?



Paper:
https://iacr.org/submit/files/slides/2024/rwc/rwc2024/1/slides.pptx

Questions?