NTNU | Norwegian University of Science and Technology

SIDE-CHANNEL ATTACKS 2

TTM4205 – Lecture 8

Tjerand Silde

19.09.2023

Contents

Contents

Reference Group Meeting

The reference group will meet Thursday morning. Get in touch with the reference group members if you have any feedback about the course. You can also provide feedback (anonymously) on the Piazza forum.

Contents

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 then analyze the scheme mathematically as black-box algorithms that take some (public or secret) input and give some (public or secret) output, and prove that it is 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 increase...
- 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

Contents

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.

Assumptions

In this example we assume a few things:

- the RSA primes are generated securely
- order phi is computed as $\operatorname{lcm}(p-1, q-1)$
- 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

```
# compute m = c**d mod n
def squareAndMultiply(c, d, n):
    m = c

    for i in range(len(d)):
        m = m * m % n

        if ( d[i] == 1 ):
            m = m * c % n

    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

```
# compute m = c**d mod n
def squareAndAlwaysMultiply(c, d, n):
    m, x = c, c

    for i in range(len(d)):
        m = m * m % n

        if ( d[i] == 1 ):
            m = m * c % n

        else:
            x = m * c % n

    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

```
# 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

```
# compute m = c**d mod n
# we have e*d = 1 mod phi
def randMontgomeryLadder(c, e, d, phi, n):

    r1 = secrets.randbelow(n)
    r2 = squareAndMultiply(r1, e, n)
    r1Inv = MontgomeryLadder(r1, phi-1, n)

    m1 = c * r2 % n
    m2 = m1 * m1 % 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

    m1 = m1*r1Inv % n
    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

```
# compute m = c**d mod n
# we have e*d = 1 mod phi
def randRandMontgomeryLadder(c, e, d, phi, n, t):

    r1 = secrets.randbelow(n)
    r2 = squareAndMultiply(r1, e, n)
    r1Inv = MontgomeryLadder(r1, phi-1, n)

    r = secrets.randbelow(t)
    # get dNew = d + r * phi
    dNew = convert(d, r, phi)

    m1 = c * r2 % n
    m2 = m1 * m1 % n

    for i in range(len(dNew)):

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

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

    m1 = m1*r1Inv % n
    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

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.

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.

Arm MUL

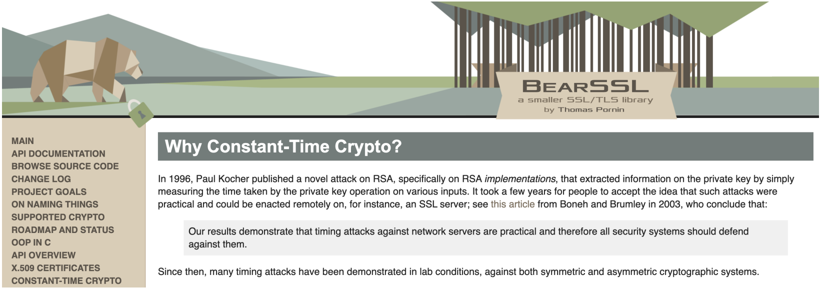
CPU type	32→32	32→64	MUL31	64→64
ARM7T (A32)	N	N	Y	S-
ARM7T (Thumb)	N	S-	S-	S-
ARM9T (A32)	N	N	Y	S-
ARM9T (Thumb)	N	S-	S-	S-
ARM9E	Y	Y	Y	S+
ARM10E	Y	Y	Y	S+
ARM11	Y	Y	Y	S+
Cortex-A (A32)	Y	Y	Y	S+
Cortex-A (A64)	?	?	?	?
Cortex-R (A32)	Y	Y	Y	S+
Cortex-M0/M0+/M1	Y	S-	S-	S-
Cortex-M3	Y	N	N	S-
Cortex-M4	Y	Y	Y	S+

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

Modular Montgomery multiplication

```
void
br_i31_montymul(uint32_t *d, const uint32_t *x, const uint32_t *y,
	const uint32_t *m, uint32_t m0i)
{
	size_t len, len4, u, v;
	uint64_t dh;

	len = (m[0] + 31) >> 5;
	len4 = len & ~(size_t)3;
	br_i32_zero(d, m[0]);
	dh = 0;
	for (u = 0; u < len; u ++) {
		uint32_t f, xu;
		uint64_t r, zh;

		xu = x[u + 1];
		f = MUL31_lo((d[1] + MUL31_lo(x[u + 1], y[1])), m0i);

		r = 0;
		for (v = 0; v < len4; v += 4) {
			uint64_t z;

			z = (uint64_t)d[v + 1] + MUL31(xu, y[v + 1])
				+ MUL31(f, m[v + 1]) + r;
			r = z >> 31;
			d[v + 0] = (uint32_t)z & 0x7FFFFFFF;
			z = (uint64_t)d[v + 2] + MUL31(xu, y[v + 2])
				+ MUL31(f, m[v + 2]) + r;
			r = z >> 31;
			d[v + 1] = (uint32_t)z & 0x7FFFFFFF;
			z = (uint64_t)d[v + 3] + MUL31(xu, y[v + 3])
				+ MUL31(f, m[v + 3]) + r;
			r = z >> 31;
			d[v + 2] = (uint32_t)z & 0x7FFFFFFF;
			z = (uint64_t)d[v + 4] + MUL31(xu, y[v + 4])
				+ MUL31(f, m[v + 4]) + r;
			r = z >> 31;
			d[v + 3] = (uint32_t)z & 0x7FFFFFFF;
		}
		for (; v < len; v ++) {
			uint64_t z;

			z = (uint64_t)d[v + 1] + MUL31(xu, y[v + 1])
				+ MUL31(f, m[v + 1]) + r;
			r = z >> 31;
			d[v] = (uint32_t)z & 0x7FFFFFFF;
		}

		zh = dh + r;
		d[len] = (uint32_t)zh & 0x7FFFFFFF;
		dh = zh >> 31;
	}

	/*
	 * We must write back the bit length because it was overwritten in
	 * the loop (not overwriting it would require a test in the loop,
	 * which would yield bigger and slower code).
	 */
	d[0] = m[0];

	/*
	 * d[] may still be greater than m[] at that point; notably, the
	 * 'dh' word may be non-zero.
	 */
	br_i31_sub(d, m, NEQ(dh, 0) | NOT(br_i31_sub(d, m, 0)));
}
```

Bear SSL


BEARSSL
a smaller SSL/TLS library
by Thomas Pornin

MAIN
API DOCUMENTATION
BROWSE SOURCE CODE
CHANGE LOG
PROJECT GOALS
ON NAMING THINGS
SUPPORTED CRYPTO
ROADMAP AND STATUS
OOP IN C
API OVERVIEW
X.509 CERTIFICATES
CONSTANT-TIME CRYPTO

Why Constant-Time Crypto?

In 1996, Paul Kocher published a novel attack on RSA, specifically on RSA *implementations*, that extracted information on the private key by simply measuring the time taken by the private key operation on various inputs. It took a few years for people to accept the idea that such attacks were practical and could be enacted remotely on, for instance, an SSL server; see this article from Boneh and Brumley in 2003, who conclude that:

> Our results demonstrate that timing attacks against network servers are practical and therefore all security systems should defend against them.

Since then, many timing attacks have been demonstrated in lab conditions, against both symmetric and asymmetric cryptographic systems.

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

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

SCA on ECC

We can essentially re-use most mechanisms for RSA in ECC.

Q: Do you see any immediate differences between the two?

SCA on ECC

We can essentially re-use most mechanisms for RSA in ECC.

A: We need to be a bit careful about the following:

- scalar multiplication must depend on curve params
- addition formulas involve inversion of secret elements
- addition formulas depends on the input points

SCA on ECC

We can essentially re-use most mechanisms for RSA in ECC.

Sol: Some possible solutions to avoid the above:

- verify points and use curve-dependent formulas
- use curves and formulas that are universal
- compute inversion in constant time (Fermat trick)
- avoid (most) inversions using projective coordinates

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

Questions?