



FROM RESEARCH TO INDUSTRY

cea tech

# Side-Channel Attacks

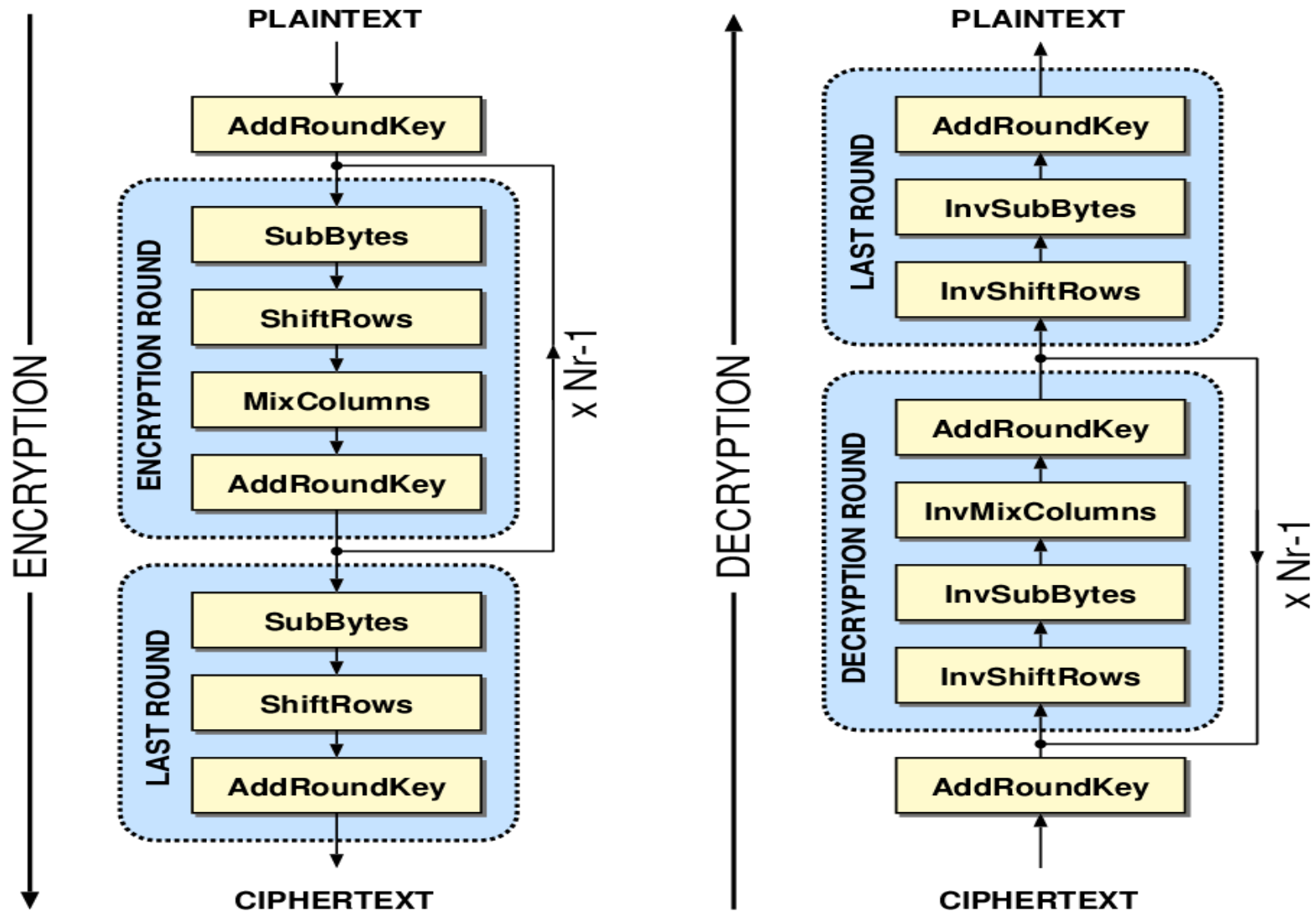
ISSISP 2017 – Gif-sur-Yvette  
2017-07-21

Damien Couroussé, CEA – LIST / LIALP; Grenoble Université Alpes  
[damien.courousse@cea.fr](mailto:damien.courousse@cea.fr)

- **Unless explicit mention at the bottom of the page, these slides are distributed under the Creative Common Attribution 3.0 License**
  - You are free:
    - to share—to copy, distribute and transmit the work
    - to remix—to adapt the work
  - under the following conditions:
    - Attribution: You must attribute the work (but not in any way that suggests that the author endorses you or your use of the work) as follows:  
“Courtesy of Damien Couroussé, CEA France”

The complete license text can be found at  
<http://creativecommons.org/licenses/by/3.0/legalcode>

# AES, TIME AFTER TIME (BUT SO USEFUL...)



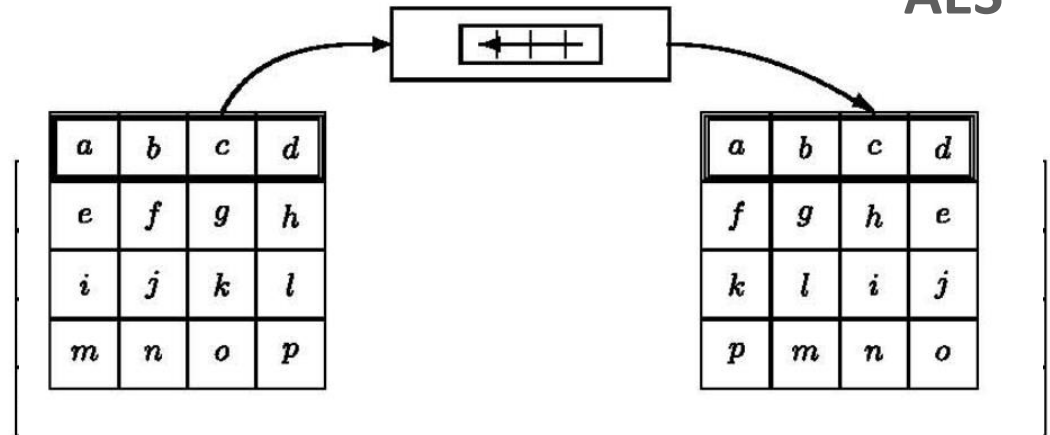
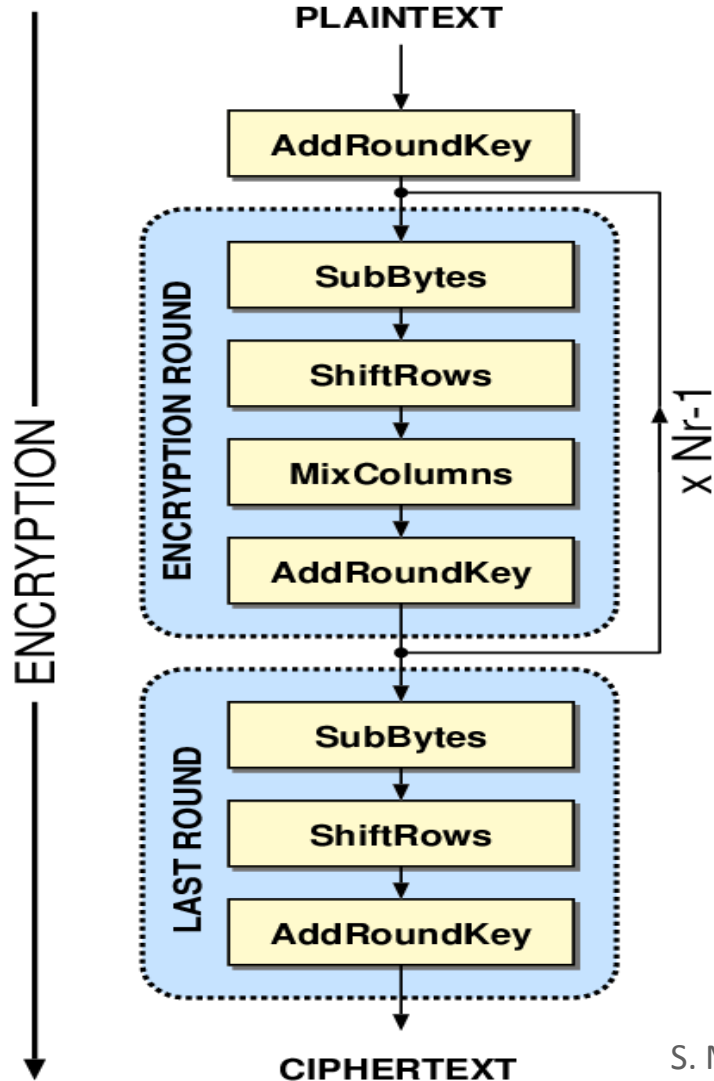


Figure B.6. ShiftRows operates on the rows of the state.

Fig

on.

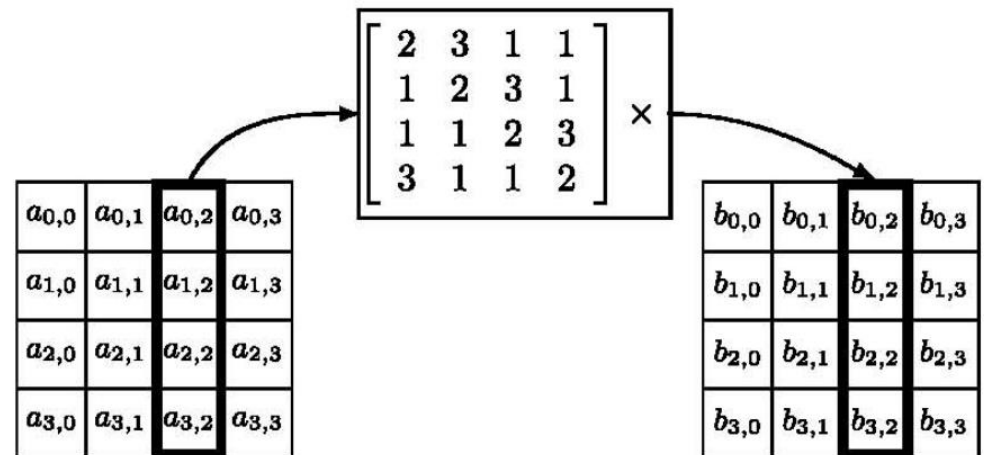


Figure B.7. MixColumns operates on the columns of the state.

S. Mangard, E. Oswald, and T. Popp, Power analysis attacks: Revealing the secrets of smart cards, vol. 31. Springer, 2007.

# BESTIARY OF EMBEDDED SYSTEMS

## ... IN NEED FOR SECURITY CAPABILITES

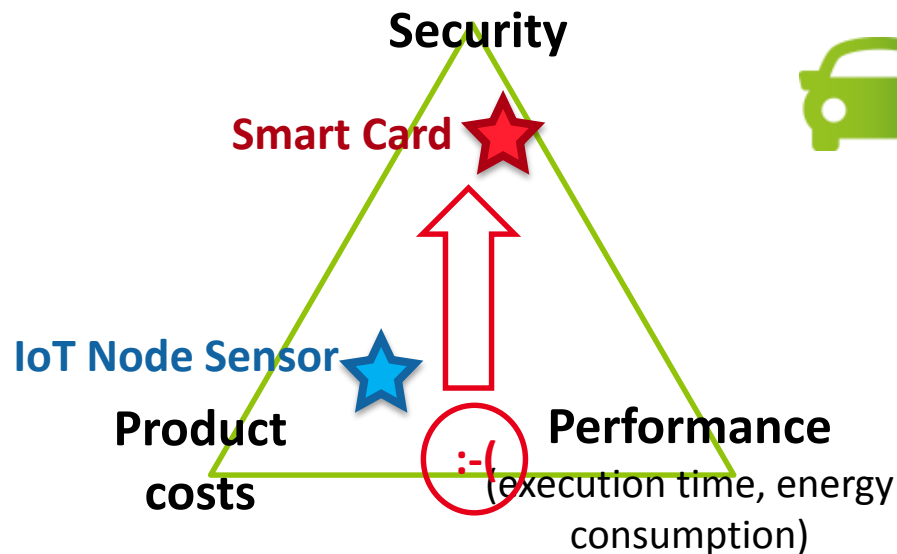
Smart Card



Secure Element inside...



... And many other things



# PHYSICAL ATTACKS: WHY ALL THE FUSS?

**Cryptography** is used to secure communications

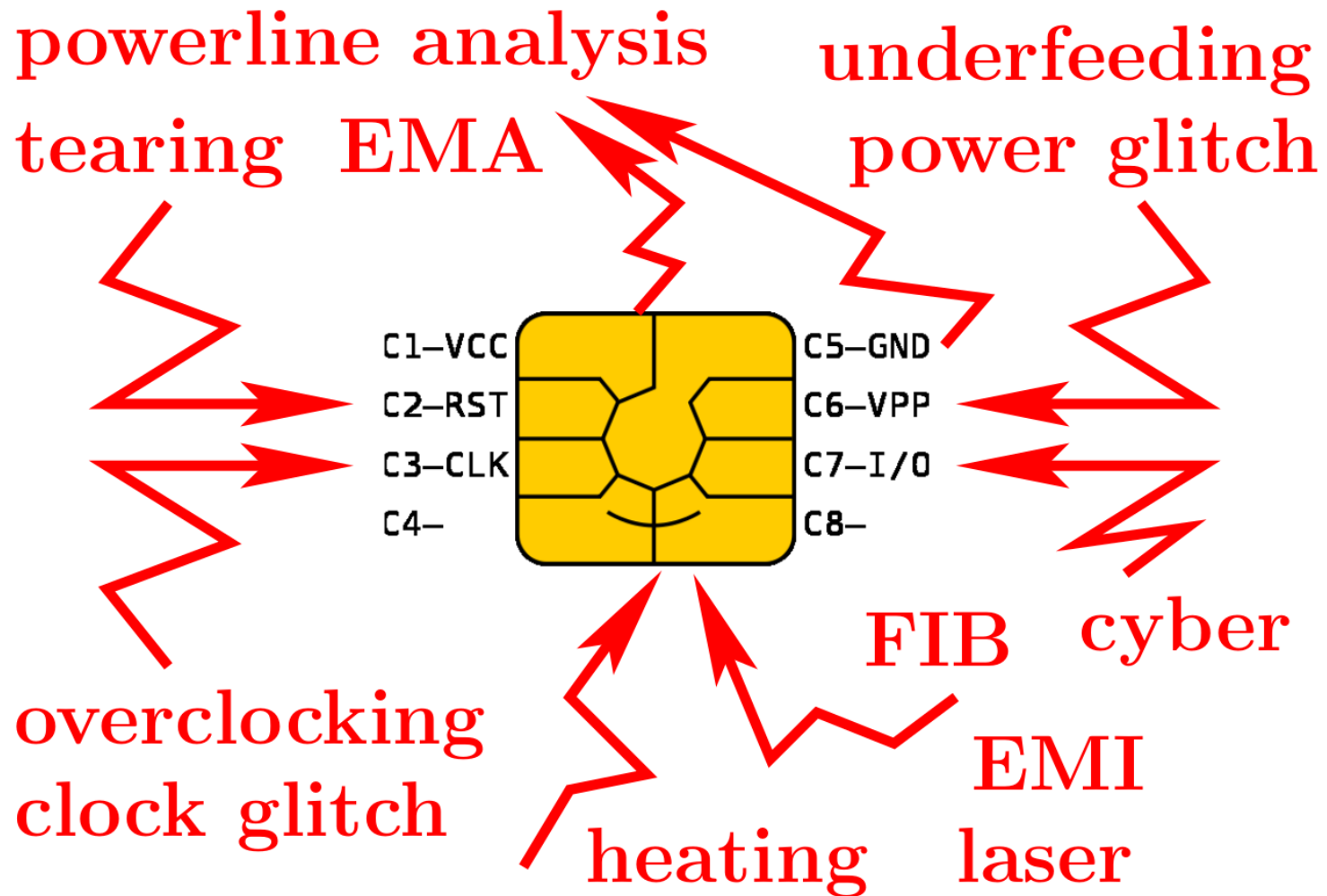
- **Encrypted data** can be safely sent over an untrusted communication channel
- Cannot recover the encrypted information without the **key**

**Cryptanalysis** studies the mathematical properties of cryptographic algorithms, and provides a “practical” confidence in security bounds.

- Security bounds are expressed in terms of **attack complexity**

**Physical attacks** are the only (effective) way to break cryptography nowadays.

- Sometimes considered as part of cryptanalysis
- But quite different research communities



Courtesy of Sylvain Guilley 2015, Télécom ParisTech - Secure-IC

An attacker proceeds in two steps:

1. Global analysis of the target, looking for potential weaknesses or known vulnerabilities – this step is not considered in the littérature.
2. Focused attack on a target

- **Cryptanalysis**

Out of the scope of this talk

- **Reverse engineering**

Hardware inspection: decapsulation, physical abrasion, chemical etching, visual inspection, etc.

Software inspection: debug, memory dumps, code analysis, etc. [see lectures past in the week]

- **Passive attacks: side-channel attacks**

Observations: electromagnetic, electrical / power, acoustic, execution time, etc. [you are here]

- **Active attacks: fault attacks**

Laser or other lights illumination, under/over-voltage, clock glitches, electromagnetic perturbations, etc. [next lecture]

- **Logical attacks**

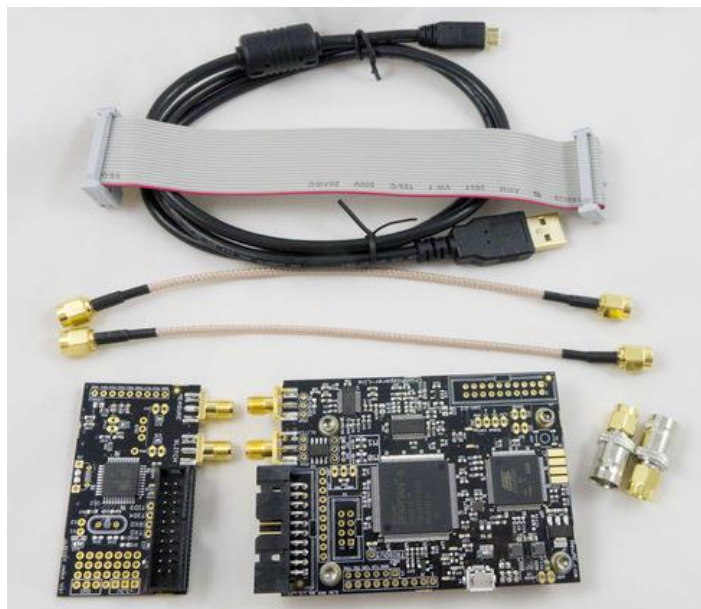
[see past lectures this weeks]

Sometimes considered as a « solved » issue in High Security products.



Physical attacks are considered (by software hackers) as not practical

- Require dedicated HW attack benches, can be quite expensive, especially for fault injection (laser benches)
- We also find low cost ones
  - E.g. *The ChipWhisperer*, starting at ~ 300€
- Require human expertise, but more than other attacks



<https://newae.com/tools/chipwhisperer>

### IoT Goes Nuclear: Creating a ZigBee Chain Reaction

*“Adjacent IoT devices will infect each other with a worm that will rapidly spread over large areas”*

- **Philips Hue Smart lamp**

- ZigBee protocol

- **Uploading malicious firmware with OTA update**

- Discovered the hex command code for OTA update
- Firmware is protected with a single global key! Using symmetric crypto (AES-CCM).

- **Attack path**

- Get access to the key → **side-channel attack with power analysis**
- Sign a malicious firmware
- Take over bulbs by: plugging a bulb, war-driving around in a car, war-flying with a drone
- Request OTA update
- The malicious firmware can request OTA update to its neighbours to spread.

### IoT Goes Nuclear: Creating a ZigBee Chain Reaction

Eyal Ronen(✉)\*, Colin OFlynn†, Adi Shamir\* and Achi-Or Weingarten\*  
PRELIMINARY DRAFT, VERSION 0.91

\*Weizmann Institute of Science, Rehovot, Israel

{[eyal.ronen](mailto:eyal.ronen@weizmann.ac.il),[adi.shamir](mailto:adi.shamir@weizmann.ac.il)}@weizmann.ac.il

†Dalhousie University, Halifax, Canada  
[coflynn@dal.ca](mailto:coflynn@dal.ca)

Other interesting read: N. Timmers and A. Spruyt, “Bypassing Secure Boot using Fault Injection,” presented at the Black Hat Europe 2016, 04-Nov-2016.

# SIDE-CHANNEL ATTACKS FOR REVERSE ENGINEERING

- Reverse-engineering from side-channel analysis
  - Even simpler on interpreters
- SCARE attacks: recovering looking tables with side-channel analysis
- FIRE attacks: using fault attacks

T. Eisenbarth, C. Paar, and B. Weghenkel, “Building a Side Channel Based Disassembler,” in Transactions on Computational Science X, Springer, Berlin, Heidelberg, 2010, pp. 78–99.

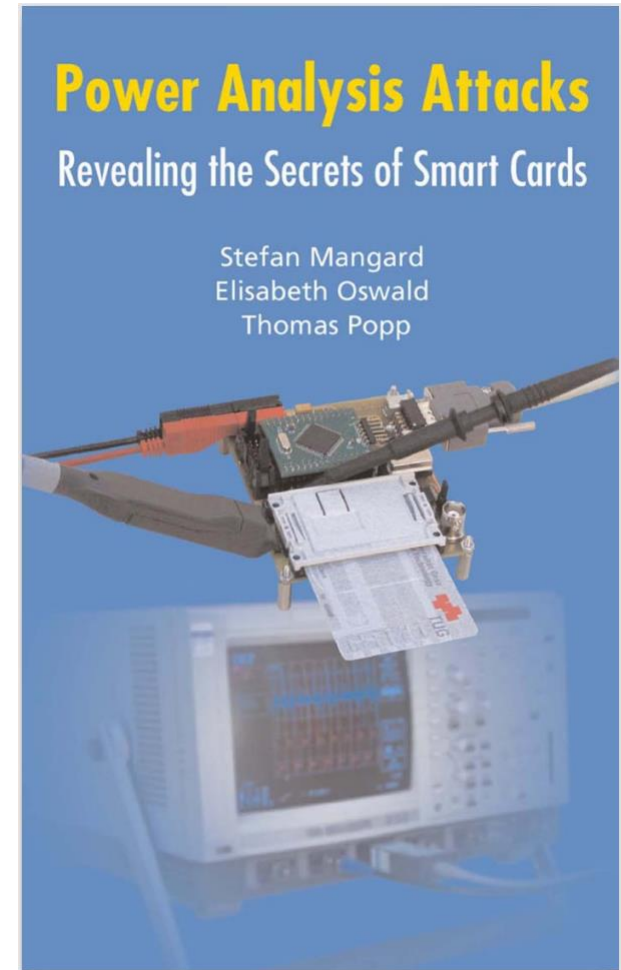
M. S. Pedro, M. Soos, and S. Guilley, “FIRE: Fault Injection for Reverse Engineering,” in Information Security Theory and Practice. Security and Privacy of Mobile Devices in Wireless Communication, 2011, pp. 280–293.

C. Clavier, “An Improved SCARE Cryptanalysis Against a Secret A3/A8 GSM Algorithm,” in Information Systems Security, 2007, pp. 143–155.

## The most comprehensive book about side-channel attacks

- Excellent introduction to side-channel attacks
- Published in 2007: does not cover recent attacks and countermeasures

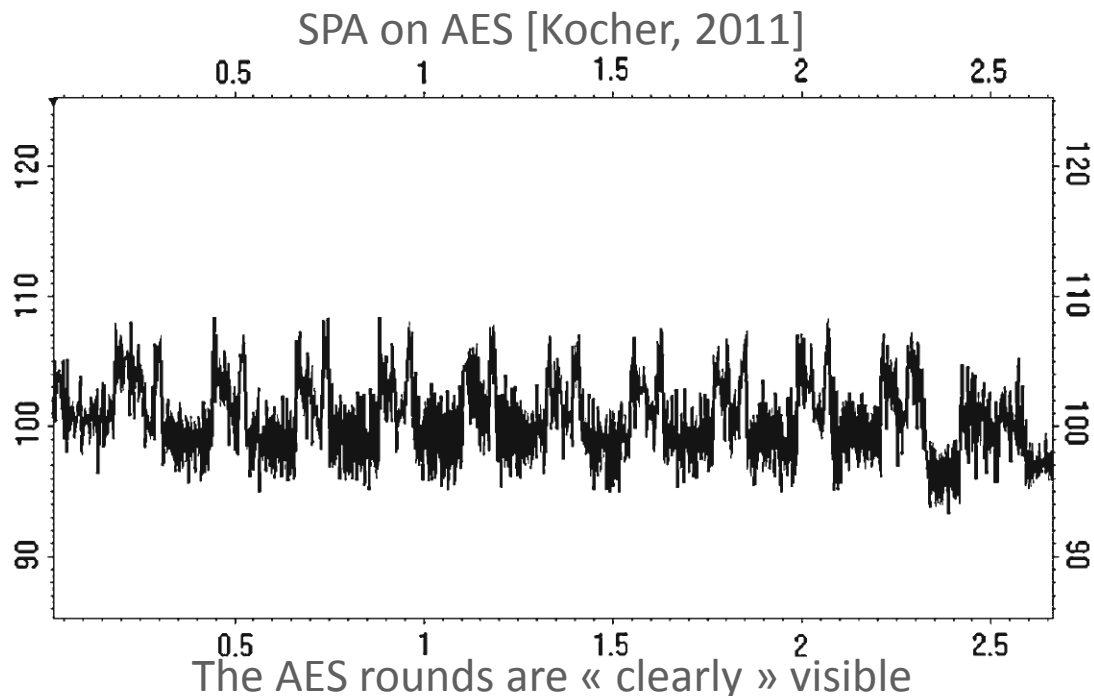
S. Mangard, E. Oswald, and T. Popp, Power analysis attacks: Revealing the secrets of smart cards, vol. 31. Springer, 2007.



## SIMPLE POWER ANALYSIS (SPA)

Direct interpretation of power consumption measurements

Extraction of information by inspection of single side-channel traces



- Nature of the algorithm
- Structure of the algorithm
  - Number of executions
  - Number of iterations
  - Number of sub-functions
  - nature of instructions executed (memory accesses...)
- Etc.

Illustration of SPA in the wild: C. O’Flynn, “A Lightbulb Worm? A teardown of the Philips Hue,” presented at the Black Hat, 2016. cf. slides ~60 to 70

P. Kocher, J. Jaffe, and B. Jun, “Differential Power Analysis,” in Advances in Cryptology — CRYPTO’ 99, vol. 1666, M. Wiener, Ed. Springer Berlin Heidelberg, 1999, pp. 388–397.

P. Kocher, J. Jaffe, B. Jun, and P. Rohatgi, “Introduction to differential power analysis,” Journal of Cryptographic Engineering, vol. 1, no. 1, pp. 5–27, 2011.

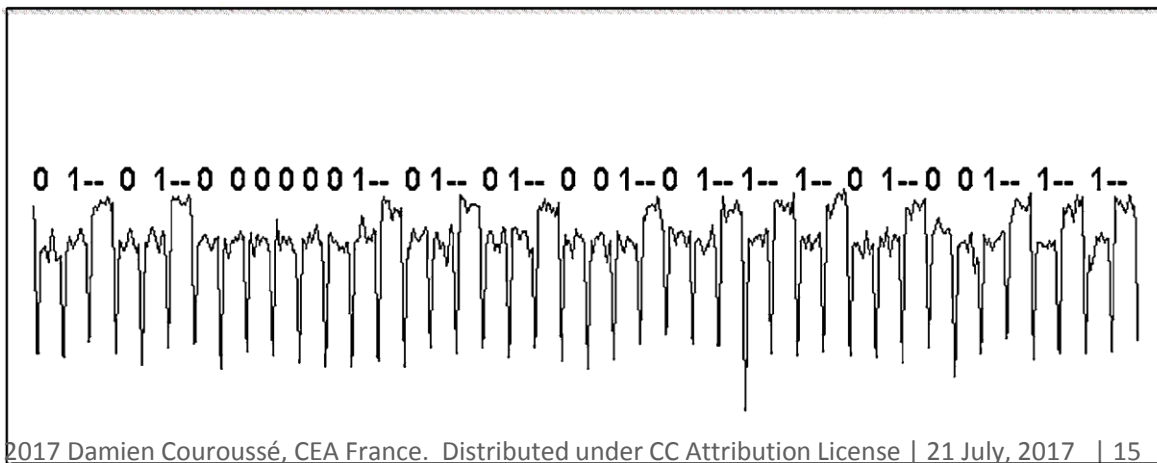
## SPA on RSA [Kocher, 2011]

```
-- Computing  $c = b^e \bmod m$   
-- Source: https://en.wikipedia.org/wiki/Modular\_exponentiation
```

```
function modular_pow(base, exponent, m)  
    if modulus = 1 then return 0  
    Assert :: (m - 1) * (m - 1) does not overflow base  
    result := 1  
    base := base mod m  
    while exponent > 0  
        if (exponent mod 2 == 1):  
            result := (result * base) mod m  
        exponent := exponent >> 1  
        base := (base * base) mod m  
    return result
```

Direct access to key contents:

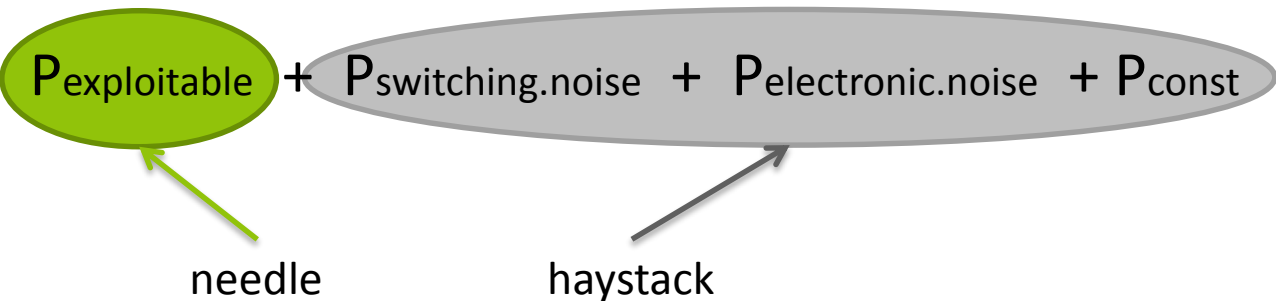
- bit 0 = square
- bit 1 = square, multiply



## Finding a needle in a haystack...

- Relationship between the different components of power consumption:

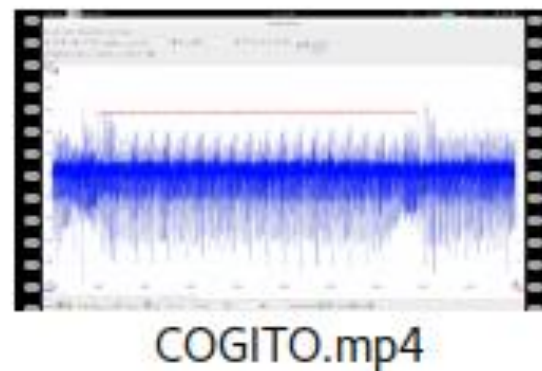
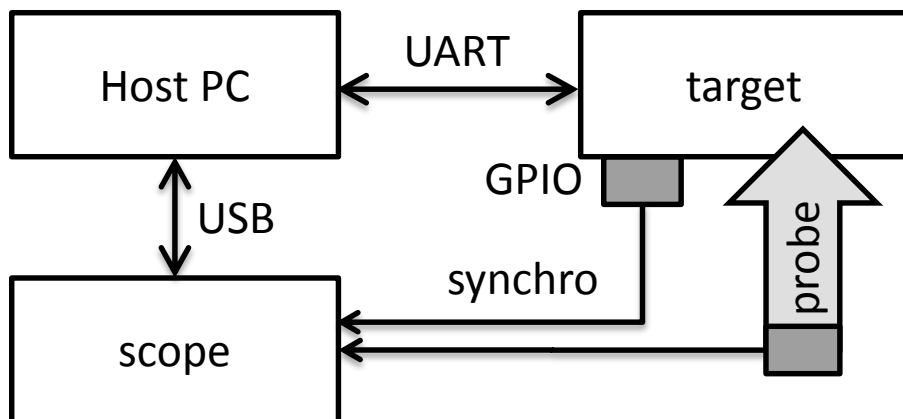
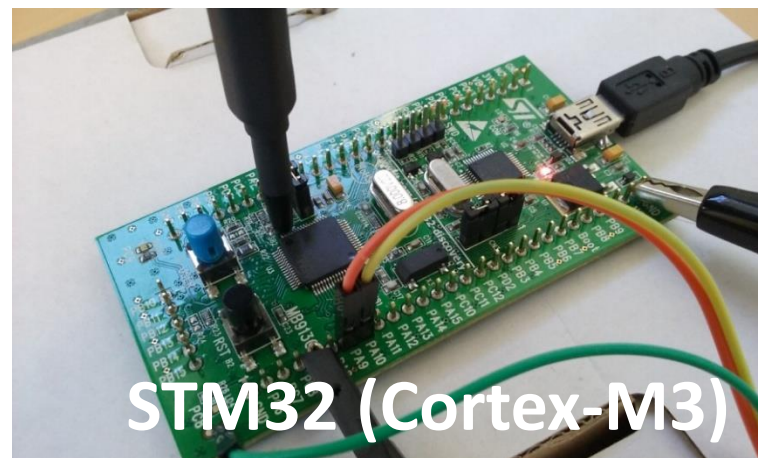
$$P_{\text{total}} = P_{\text{operations}} + P_{\text{data}} + P_{\text{noise}}$$

$$P_{\text{total}} = P_{\text{exploitable}} + P_{\text{switching.noise}} + P_{\text{electronic.noise}} + P_{\text{const}}$$


- Power signal: a static and a dynamic component.
  - Static component: power consumption of the gate states  $\rightarrow a * HW(\text{state})$
  - Dynamic component: power consumption of transitions in gate states  $\rightarrow b * HD(\text{state}[i], \text{state}[i-1])$
- Other needles & stacks
  - Electromagnetic emissions
  - Execution time
  - Chip temperature
  - Etc.

## CPA – MEASUREMENT SETUP

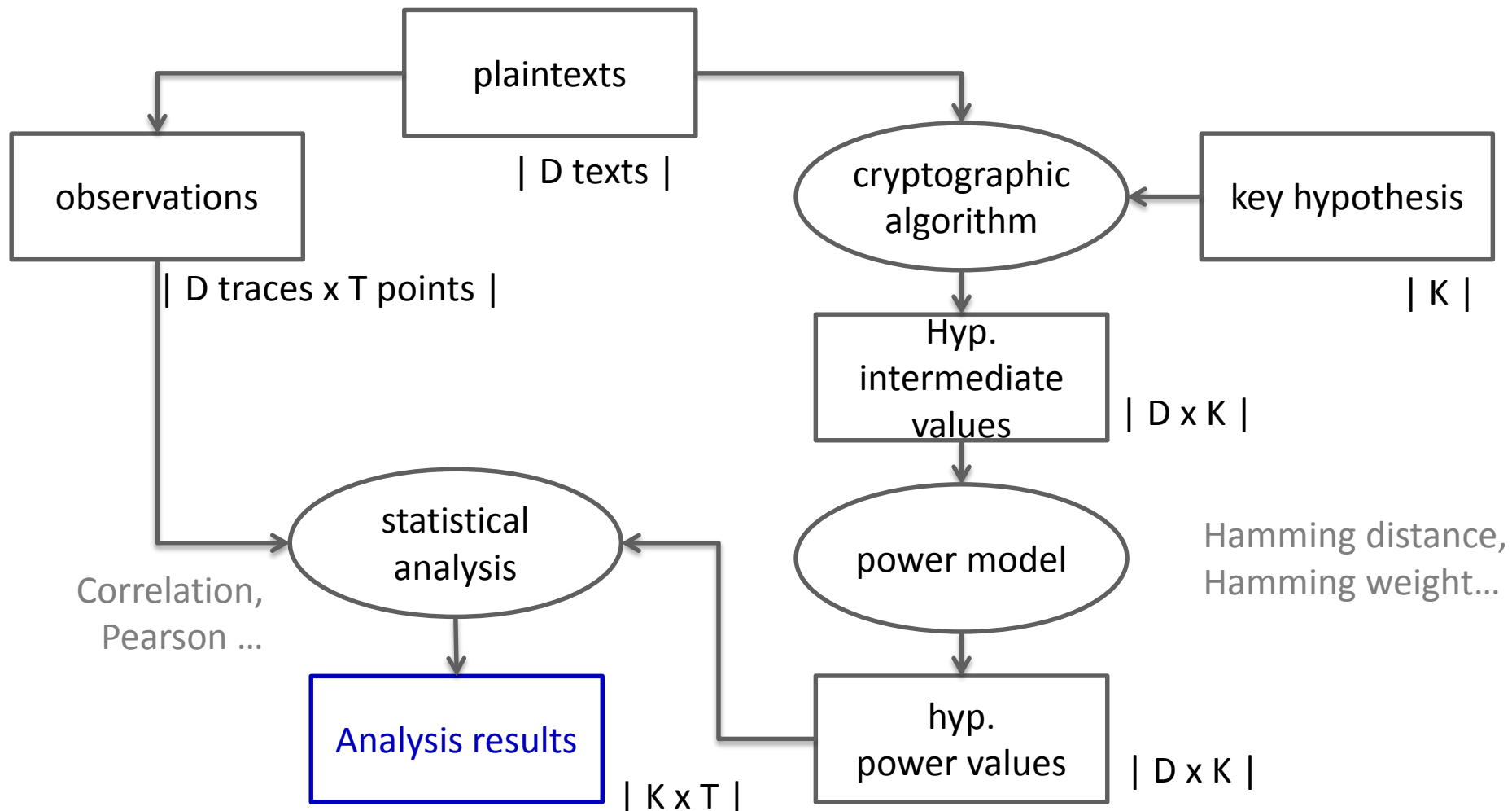
- **Target:** STM32 – ARM Cortex-M3 @ 24MHz, 128KB flash, 8KB RAM
- The AES key is fixed
- A GPIO trigger is used to facilitate the trace measurements
- The attacker either knows the plaintext or the ciphertext (public data)
- Text chosen attack:
  - Generate D random plaintexts
  - Ask the cipher text to the target
  - Record the EM trace during encryption
- **Do the computation analysis!**

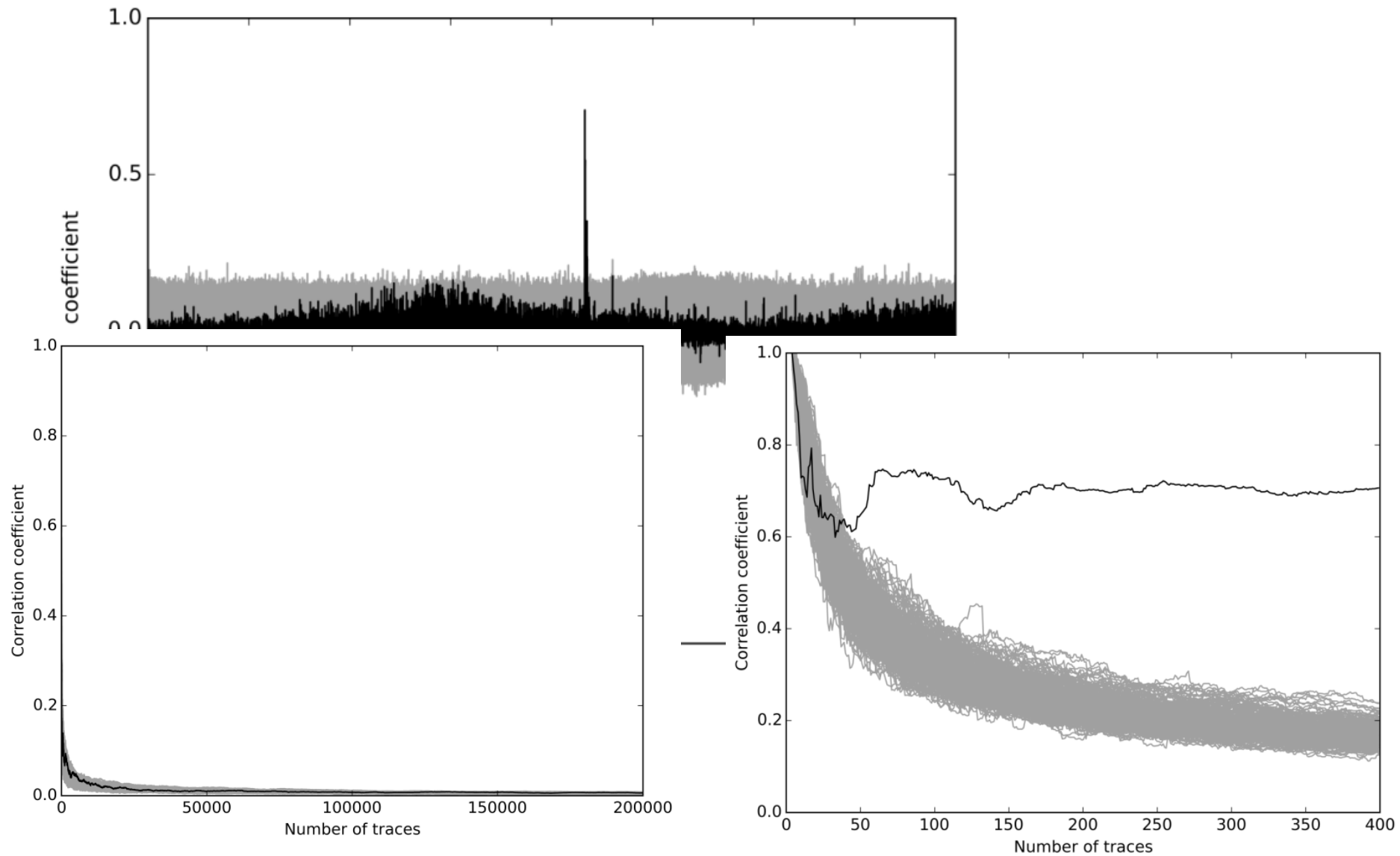




m: plaintext -> controlled by the attacker or observable

k: cipher key -> unknown to the attacker





# ESTIMATING THE SUCCESS OF AN ATTACK

**Success rate: success probability of a successful attack**

$$SR = \Pr[ A(E_{k_0}, L) = k_0 ]$$

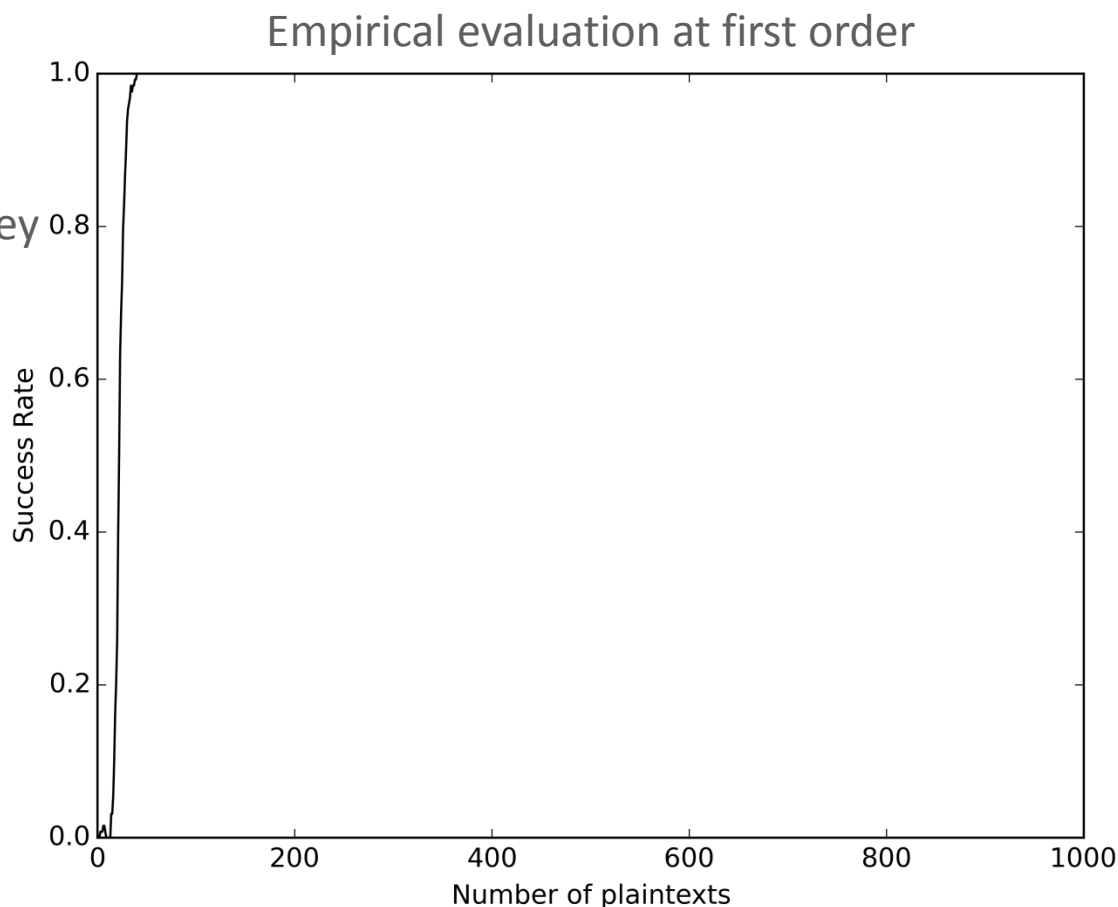
A side-channel attack

$k_0$  correct key

$E_{k_0}$  encryption with correct key

L leakage

n-order success rate?



F.-X. Standaert, T. Malkin, and M. Yung, “A Unified Framework for the Analysis of Side-Channel Key Recovery Attacks,” in Eurocrypt, 2009, vol. 5479, pp. 443–461.

- **CPA / DPA ... attacks do not constitute a security evaluation.**
- **Playing the role of the attacker is great, but the attacker**
  - is focused on a potential vulnerability
  - Follows a specific attack path
- **Starting from the previous attack, we could change**
  - The hypothetical intermediate values: output of 1st SubBytes, output of 1st AddRoundKey, input of the 10th SubBytes...
  - The power model: Hamming Weight, Hamming Distance, no power model...
  - The distinguisher: Pearson Correlation, Mutual Information...
  - There are many other attacks!
- **Our evaluation target is very “leaky” (less than 1000 traces is enough)**
  - Unprotected components executed on more complex targets (i.e. ARM Cortex A9) will require 100.000 to  $10^6$  traces.
  - What about attacking a counter-measure in this case?
- **As a security designer, you need to cover all the possible attack passes**

## TLVA: Test Leakage Vector Assessment

- Exploit Welch's t-test to assess the amount of information leakage
- Extract two populations of side-channel observations (traces)
- Test the null hypothesis: the two populations are not statistically distinguishable → no information leakage

$$t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{s_0^2}{n_0} + \frac{s_1^2}{n_1}}},$$

$t > 4.5 \rightarrow$  confidence of 99.999% that the null hypothesis is rejected



G. Goodwill, B. Jun, J. Jaffe, and P. Rohatgi, “A testing methodology for side-channel resistance validation,” in NIST non-invasive attack testing workshop, 2011.

D. B. Roy, S. Bhasin, S. Guilley, A. Heuser, S. Patranabis, and D. Mukhopadhyay, “Leak Me If You Can: Does TVLA Reveal Success Rate?,” 1152, 2016.

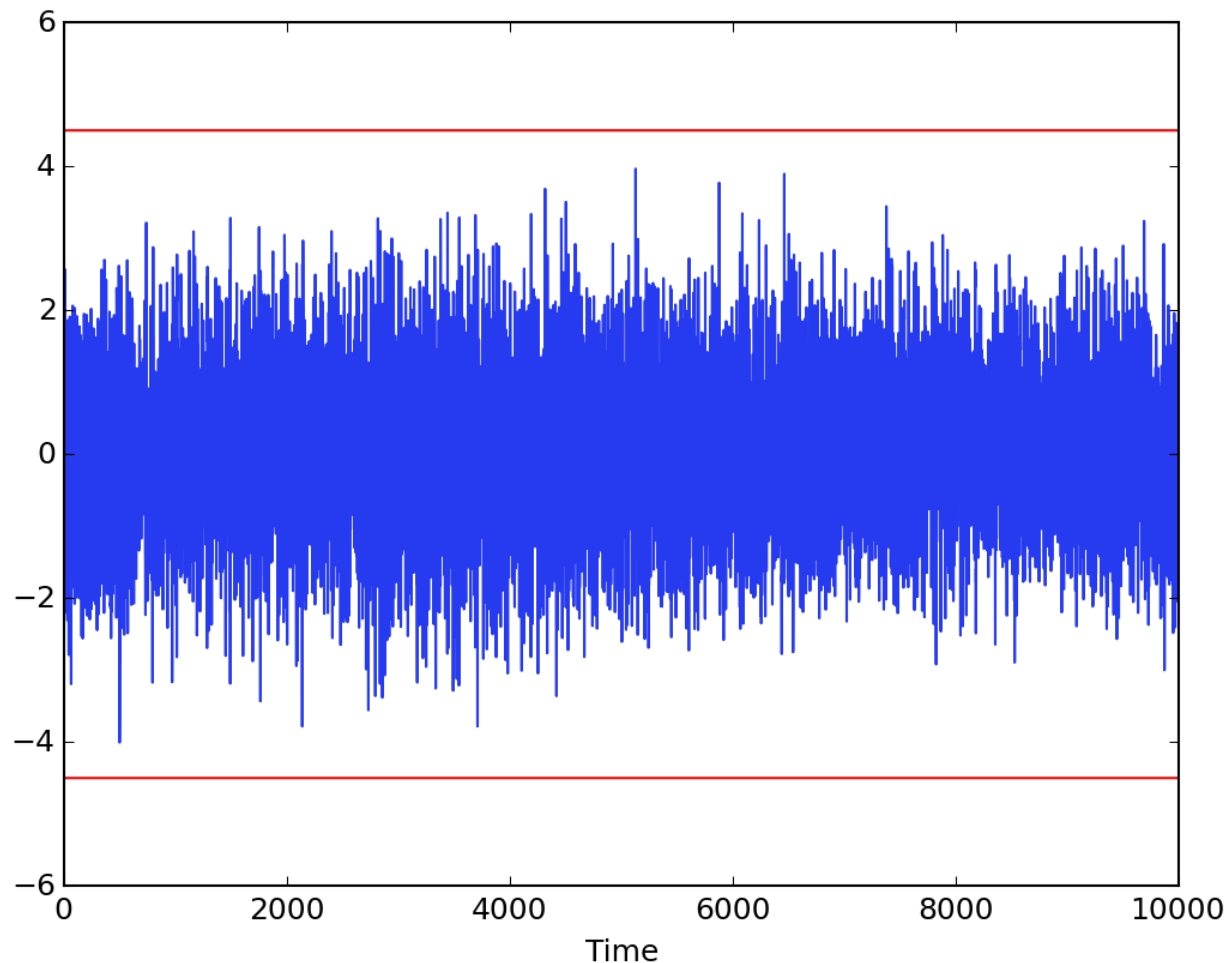
T. Schneider and A. Moradi, “Leakage Assessment Methodology - a clear roadmap for side-channel evaluations,” 207, 2015.

## TLVA: Test Leakage Vector Assessment

- Exploit
- Extract
- Test the distinguishing

$$t = \frac{\mu}{\sqrt{\frac{s}{n}}}$$

T-TEST



hypothesis

G. Goodwill, B. Schneier, and S. J. Stanger, "Non-invasive a

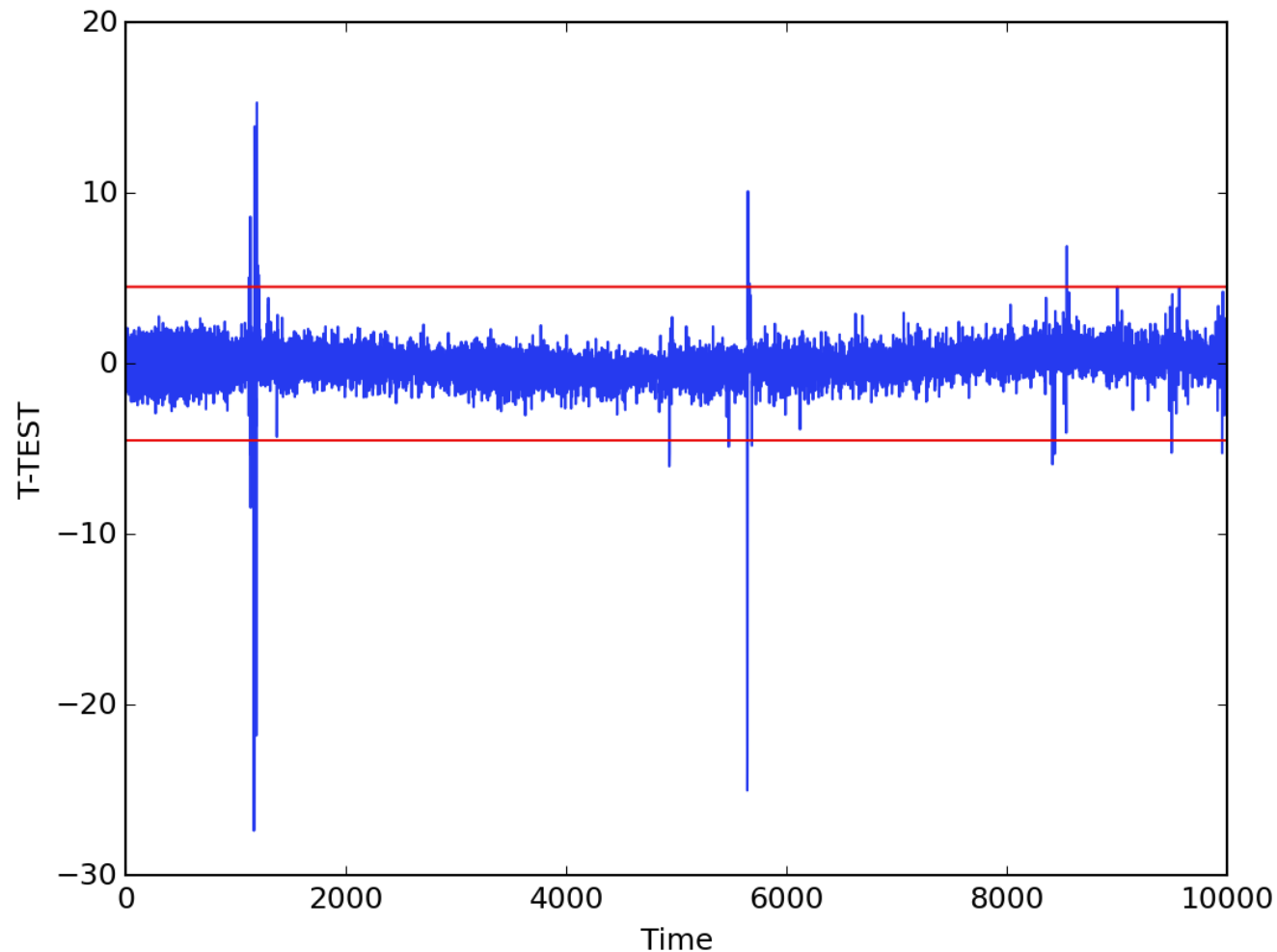
D. B. Roy, S. Bhattacharya, S. Guin, A. Heuser, S. P. Palanisami, and D. Mukhopadhyay, "Leak Me If You Can: Does TVLA Reveal Success Rate?," 1152, 2016.

T. Schneider and A. Moradi, "Leakage Assessment Methodology: a clear roadmap for side-channel evaluations," 2015.

$$Q_0 = \{T_i \mid \text{target bit}(D_i) = 0\}, \quad Q_1 = \{T_i \mid \text{target bit}(D_i) = 1\}.$$

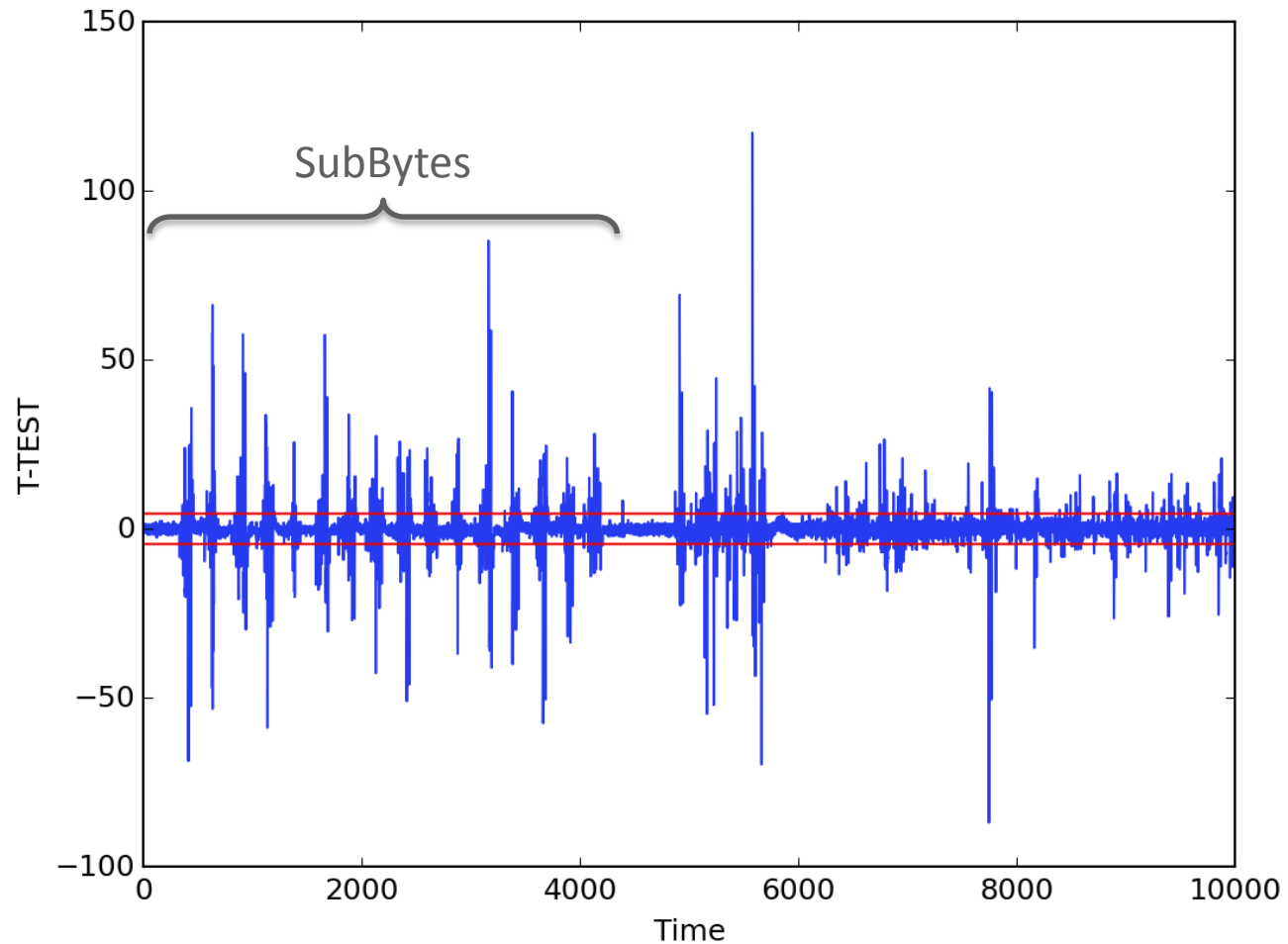
$$Q_0 = \{T_i \mid \text{target byte}(D_i) = x\}, \quad Q_1 = \{T_i \mid \text{target byte}(D_i) \neq x\}.$$

Number of  
measurements for a  
security evaluation?



*Q0: fixed input plaintext*

*Q1: random input plaintext*





# **COUNTER-MEASURES AGAINST SIDE-CHANNEL ATTACKS**

**MASKING    &    MASKING**

## MASKING

In a masked implementation, **each intermediate value  $v$  is concealed** by a random value  $m$  that is called mask:  
 **$Vm = v * m$** . The mask  $m$  is generated internally, i.e. inside the cryptographic device, and varies from execution to execution. Hence, it is not known by the attacker.

[DPA book]



- **Boolean masking:** operator  $*$  is *xor*
- **Arithmetic masking:** operator  $*$  is the modular addition or the modular multiplication

Objective: **each masked variable is statistically independent of the secret  $v$** .  
A (first-order) CPA attack can recover a (first-order) masked variable, but this knowledge is not sufficient to recover the secret value.

Masking countermeasures are applied at the **algorithmic level**.

Our following discussions will be based on the parallel implementation of a masking scheme such as described in [2]. More precisely, we will consider the simplest example where all the shares are in  $\text{GF}(2)$  (generalizations to other fields follow naturally). In this setting, we have a sensitive variable  $x$  that is split into  $m$  shares such that  $x = x_1 \oplus x_2 \oplus \dots \oplus x_m$ , with  $\oplus$  the bitwise XOR. The first  $m - 1$  shares are picked up uniformly at random:  $(x_1, x_2, \dots, x_{m-1}) \xleftarrow{\mathcal{R}} \{0, 1\}$ , and the last one is computed as  $x_m = x \oplus x_1 \oplus x_2 \oplus \dots \oplus x_{m-1}$ .

Denoting the vector of shares  $(x_1, x_2, \dots, x_m)$  as  $\bar{x}$ , we will consider an adversary who observes a single leakage sample corresponding to the parallel manipulation of these shares. A simple model for this setting is to assume this sample to be a linear combination of the shares, namely:

$$L_1(\bar{x}) = \left( \sum_{i=1}^m \alpha_i \cdot x_i \right) + N,$$

F.-X. Standaert, “How (not) to Use Welch’s T-test in Side-Channel Security Evaluations,” 138, 2017.

The goal of hiding countermeasures is to make the **power consumption** of cryptographic devices **independent of the intermediate values** and **independent of the operations** that are performed. There are essentially two approaches to achieve this independence.

1. the **power consumption is random**.
2. **consume an equal amount of power** for all operations and for all data values.

[DPA book]

Hiding countermeasures aim at **breaking the observable relation** between **the algorithm** (operations and intermediate variables) and **observations**.



**Information leakage:** information related to secret data and secret operations “sneaks” outside of the secured component (via a side channel)

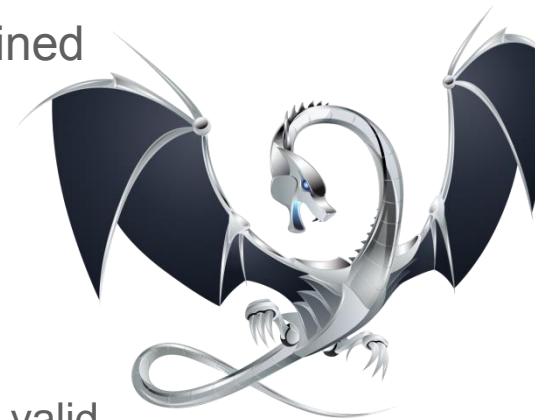
**Hiding:** “reducing the SNR”, where

- Signal -> information leakage
- Noise -> everything else
- Temporal dispersion: spread leakage at different computation times
  - Shuffle independent operations
  - Insert «dummy» operations to randomly delay the secret computation
- Spatial dispersion:
  - Move the leaky computation at different places in the circuit
    - E.g. use different registers
  - Modify the “appearance” of information leakage
    - E.g. use different operations

**In practice, a secured product combines masking and hiding countermeasures.**

# **STANDARD COMPILERS AND SECURITY**

- **Duties: assurance of functional equivalence between source code and machine code**
  - “functional” / “functionality” is usually not precisely defined
    - Side effects?
    - Determinism of time behaviour? (real time execution)
    - Lazy evaluation?
  - No formal assurance
    - Except few works, such as CompCert
  - Correctness by construction?
    - The source code written by the developer is not always valid
- **Objectives: optimise one or several performance criteria**
  - Execution time
  - Resources: e.g. memory consumption
  - Energy consumption, power consumption
  - There is no complete criterion for optimality, and no convergence
    - Nature of the algorithm used
    - Relation to architecture / micro-architecture
    - Data...



- **Rights**

- Reorganise contents of the target program, as long as program semantics is preserved
  - Machine instructions, basic blocs
- Select the best translation for a source code operation / instruction
- Remove parts of the program, as long as the program functionality is considered to be preserved (i.e. the computation does not participate in producing the program results)

- **Some classical optimisation passes:**

- *dead code elimination*
- *global value numbering*
- *common-subexpression elimination*
- *strength reduction*
- *loop strength reduction, loop simplification, loop-invariant code motion*

- **LLVM's Analysis and Transform Passes, the 2016/06/30**

- 40 analysis passes
- 56 transformation/optimisation passes
- 10 utility passes
- ... backends, etc.

Will break your security mechanisms



# **USE OF A STANDARD COMPILER, IMPACT ON SECURITY**

# INSERTION OF DUMMY INSTRUCTIONS

- Inserting a static procedure for desynchronisation

```
/* subBytes
 * Table Lookup
 */
void subBytes_f(void)
{
    int i;
```

```
    for(i = 0; i<16; i+=4)
```

```
    {
        CORON();
        state[i+0] = sbox[ state[i+0] ];
        state[i+1] = sbox[ state[i+1] ];
        state[i+2] = sbox[ state[i+2] ];
        state[i+3] = sbox[ state[i+3] ];
    }
```

```
}
```

```
void noiseCoron(void)
{
```

```
    size_t i;
    if(nbIt_Coron == N) {
        genNoiseCoron();
    }
```

```
    /* random delay */
```

```
    i = 0;
    while(i < table_d[nbIt_Coron]) {
        i++;
    }
```

```
    nbIt_Coron++;
```

```
}
```

- Also possible (even better) with a timer and an interrupt handler

Coron, J. S., & Kizhvatov, I. An efficient method for random delay generation in embedded software. In Cryptographic Hardware and Embedded Systems-CHES 2009 (pp. 156-170). Springer (2009).

Coron, J.S., Kizhvatov, I. Analysis and improvement of the random delay countermeasure of CHES 2009. In: CHES. pp. 95–109. Springer (2010).



# INSERTION OF DUMMY INSTRUCTIONS

Compiled with -Os:

Dump of assembler code for function noiseCoron:

```
void noiseCoron(void)
{
    size_t i;
    if(nbIt_Coron == N) {
        genNoiseCoron();
    }

    /* random delay */
    i = 0;
    while(i < table_d[nbIt_Coron]) {
        i++;
    }

    nbIt_Coron++;
}
```

```
0x0000859c <+0>:    push        {r4, lr}
0x000085a0 <+4>:    ldr         r4, [pc, #28] ; <noiseCoron+40>
0x000085a4 <+8>:    ldr         r3, [r4]      ; r3 ← nbIt_coron
0x000085a8 <+12>:   cmp         r3, #160      ; nbIt_coron ?= N
0x000085ac <+16>:   bne         0x85b4 <noiseCoron+24>
0x000085b0 <+20>:   bl         0x8524 <genNoiseCoron>
0x000085b4 <+24>:   ldr         r3, [r4]
0x000085b8 <+28>:   add         r3, r3, #1 ; nbIt_coron++
0x000085bc <+32>:   str         r3, [r4]
0x000085c0 <+36>:   pop         {r4, pc}
0x000085c4 <+40>:   andeq       r0, r1, r0, lsr r8
```

End of assembler dump.

???



# INSERTION OF DUMMY INSTRUCTIONS

## Compiled with -Os:

Dump of assembler code for function noiseCoron:

```
void noiseCoron(void)
{
    size_t i;
    if(nbIt_Coron == N) {
        genNoiseCoron();
    }

    /* random delay */
    i = 0;
    while(i < table_d[nbIt_Coron]) {
        i++;
        asm("nop;");
    }

    nbIt_Coron++;
}
```

```
0x0000859c <+0>:      push      {r4, lr}
0x000085a0 <+4>:      ldr        r4, [pc, #60]      ; <noiseCoron+72>
0x000085a4 <+8>:      ldr        r3, [r4]
0x000085a8 <+12>:     cmp        r3, #160      ; nbIt_coron ?= N
0x000085ac <+16>:     bne        0x85b4 <noiseCoron+24>
0x000085b0 <+20>:     bl         0x8524 <genNoiseCoron>
0x000085b4 <+24>:     ldr        r3, [pc, #44]      ; <noiseCoron+76>
0x000085b8 <+28>:     ldr        r2, [r4]
0x000085bc <+32>:     ldr        r1, [r3, r2, lsl #2]
0x000085c0 <+36>:     mov        r3, #0          ; i ← 0
0x000085c4 <+40>:     cmp        r3, r1          ; i ?= nbIt_Coron
0x000085c8 <+44>:     beq        0x85d8 <noiseCoron+60>
0x000085cc <+48>:     add        r3, r3, #1      ; i ← i+1
0x000085d0 <+52>:     nop
0x000085d4 <+56>:     b         0x85c4 <noiseCoron+40>
0x000085d8 <+60>:     add        r2, r2, #1      ; nbIt_Coron++
0x000085dc <+64>:     str        r2, [r4]
0x000085e0 <+68>:     pop        {r4, pc}
0x000085e4 <+72>:     andeq     r0, r1, r4, asr r8
0x000085e8 <+76>:     andeq     r0, r1, r12, asr r8
```

End of assembler dump.



# INSERTION OF DUMMY INSTRUCTIONS

## Compiled with -Os:

Dump of assembler code for function noiseCoron:

```
void noiseCoron(void)
{
    size_t i;
    if(nbIt_Coron == N) {
        genNoiseCoron();
    }

    /* random delay */
    i = 0;
    while(i < table_d[nbIt_Coron]) {
        i++;
        asm("");
    }

    nbIt_Coron++;
}
```

```
0x0000859c <+0>:      push      {r4, lr}
0x000085a0 <+4>:      ldr        r4, [pc, #56] ; <noiseCoron+68>
0x000085a4 <+8>:      ldr        r3, [r4]
0x000085a8 <+12>:     cmp        r3, #160      ; 0xa0
0x000085ac <+16>:     bne        0x85b4 <noiseCoron+24>
0x000085b0 <+20>:     bl         0x8524 <genNoiseCoron>
0x000085b4 <+24>:     ldr        r3, [pc, #40] ; <noiseCoron+72>
0x000085b8 <+28>:     ldr        r2, [r4]
0x000085bc <+32>:     ldr        r1, [r3, r2, lsl #2]
0x000085c0 <+36>:     mov        r3, #0
0x000085c4 <+40>:     cmp        r3, r1
0x000085c8 <+44>:     beq        0x85d4 <noiseCoron+56>
0x000085cc <+48>:     add        r3, r3, #1
0x000085d0 <+52>:     b         0x85c4 <noiseCoron+40>
0x000085d4 <+56>:     add        r2, r2, #1
0x000085d8 <+60>:     str        r2, [r4]
0x000085dc <+64>:     pop        {r4, pc}
0x000085e0 <+68>:     andeq     r0, r1, r0, asr r8
0x000085e4 <+72>:     andeq     r0, r1, r8, asr r8
```

End of assembler dump.



# RANDOM PRECHARGING

- Protection against power analysis using a Hamming Distance model
- Example: Leakage on value  $v$  is charged in memory or in a register:

#1  $\text{insn } k$   
 $\text{mem } \overleftarrow{-} v$

**Leakage:  $\text{HD}(v, k)$**

#2  $\text{insn } k$   
 $\text{reg } \overleftarrow{-} v$

- Random precharging: the variable assignment is preceded by an assignment using a mask  $m$ , unknown to the attacker:

#1  $\text{insn } k$   
 $\text{mem } \overleftarrow{-} m$   
 $\text{mem } <- v$

**Leakage:**

**$\text{HD}(v, m) = \text{HW}(v \oplus m)$**

#2  $\text{insn } k$   
 $\text{reg } \overleftarrow{-} m$   
 $\text{reg } <- v$

```
#define SBOX_SIZE 256
uint8_t sbox[SBOX_SIZE];

#define STATE_SIZE 16
uint8_t state[STATE_SIZE];

/* subBytes, table Lookup */
void subBytes(void)
{
    size_t i;
    for(i = 0; i < SBOX_SIZE; i++) {
        state[i] = sbox[state[i]];
    }
}
```

Compiled with -Os:

```
0x0000 <+0>: mov r3, #0
0x0004 <+4>: ldr r2, [pc, #28] ; 0x28 <subBytes+40>
0x0008 <+8>: ldr r0, [pc, #28] ; 0x2c <subBytes+44>
0x000c <+12>: ldrb r1, [r3, r2]
0x0010 <+16>: ldrb r1, [r0, r1]
0x0014 <+20>: strb r1, [r3, r2] ; leaky instruction
0x0018 <+24>: add r3, r3, #1
0x001c <+28>: cmp r3, #16
0x0020 <+32>: bne 0xc <subBytes+12>
0x0024 <+36>: bx lr
0x0028 <+40>: andeq r0, r0, r0
0x002c <+44>: andeq r0, r0, r0
```

```
#define SBOX_SIZE    256
uint8_t sbox[SBOX_SIZE];

#define STATE_SIZE    16
uint8_t state[STATE_SIZE];

/* subBytes, table Lookup */
void subBytes(void)
{
    size_t i;
    uint8_t mask, tmp_state;

    for(i = 0; i<SBOX_SIZE; i++) {
        tmp_state = state[i];
        mask = rand() & 0xFF;

        state[i] = mask;
        state[i] = sbox[tmp_state];
    }
}
```

Compiled with -Os:

```
0x0000 <+0>:  push {r4, r5, r6, r7, r8, lr}
0x0004 <+4>:  mov r4, #0
0x0008 <+8>:  ldr r5, [pc, #36] ; <subBytes+52>
0x000c <+12>: ldr r7, [pc, #36] ; <subBytes+56>
0x0010 <+16>: ldrb r6, [r4, r5]
??? 0x0014 <+20>: bl 0x14 <subBytes+20>
0x0018 <+24>: ldrb r3, [r7, r6]
0x001c <+28>: strb r3, [r4, r5]
0x0020 <+32>: add r4, r4, #1
0x0024 <+36>: cmp r4, #16
0x0028 <+40>: bne 0x10 <subBytes+16>
0x002c <+44>: pop {r4, r5, r6, r7, r8, lr}
0x0030 <+48>: bx lr
0x0034 <+52>: andeq r0, r0, r0
0x0038 <+56>: andeq r0, r0, r0
```



## RANDOM PRECHARGING

```

#define SBOX_SIZE    256
uint8_t sbox[SBOX_SIZE];

#define STATE_SIZE    16
uint8_t volatile state[STATE_SIZE];

/* subBytes, table Lookup */
void subBytes(void)
{
    size_t i;
    uint8_t mask, tmp_state;

    for(i = 0; i<SBOX_SIZE; i++) {
        tmp_state = state[i];
        mask = rand() & 0xFF;

        state[i] = mask;
        state[i] = sbox[tmp_state];
    }
}

```

Compiled with -Os:

```

0x0000 <+0>: push {r4, r5, r6, r7, r8, lr}
0x0004 <+4>: mov r4, #0
0x0008 <+8>: ldr r5, [pc, #48] ; <subBytes+64>
0x000c <+12>: ldr r7, [pc, #48] ; <subBytes+68>
0x0010 <+16>: ldrb r6, [r5, r4]
0x0014 <+20>: bl 0x14 <subBytes+20>
0x0018 <+24>: and r6, r6, #255 ; 0xff
0x001c <+28>: ldrb r3, [r7, r6]
0x0020 <+32>: and r0, r0, #15
0x0024 <+36>: strb r0, [r5, r4]
0x0028 <+40>: strb r3, [r5, r4]
0x002c <+44>: add r4, r4, #1
0x0030 <+48>: cmp r4, #16
0x0034 <+52>: bne 0x10 <subBytes+16>
0x0038 <+56>: pop {r4, r5, r6, r7, r8, lr}
0x003c <+60>: bx lr
0x0040 <+64>: andeq r0, r0, r0
0x0044 <+68>: andeq r0, r0, r0

```





```
#define SBOX_SIZE 256
uint8_t sbox[SBOX_SIZE];

#define STATE_SIZE 16
uint8_t volatile state[STATE_SIZE];

void subBytes(void)
{
    size_t i;
    uint8_t mask, tmp_state;

    for(i = 0; i<SBOX_SIZE; i++) {
        tmp_state = state[i];
        mask = rand() & 0x000F;

        state[i] = mask;
        state[i] = sbox[tmp_state];
    }
}
```

Compiled with -O1:

```
0x0000 <+0>: push {r4, r5, r6, r7, r8, lr}
0x0004 <+4>: mov r4, #0
0x0008 <+8>: ldr r6, [pc, #48] ; <subBytes+64>
0x000c <+12>: ldr r7, [pc, #48] ; <subBytes+68>
0x0010 <+16>: ldrb r5, [r6, r4]
0x0014 <+20>: and r5, r5, #255 ; 0xff
0x0018 <+24>: bl 0x18 <subBytes+24>
0x001c <+28>: and r0, r0, #15
0x0020 <+32>: strb r0, [r6, r4]
0x0024 <+36>: ldrb r3, [r7, r5]
0x0028 <+40>: strb r3, [r6, r4]
0x002c <+44>: add r4, r4, #1
0x0030 <+48>: cmp r4, #16
0x0034 <+52>: bne 0x10 <subBytes+16>
0x0038 <+56>: pop {r4, r5, r6, r7, r8, lr}
0x003c <+60>: bx lr
0x0040 <+64>: andeq r0, r0, r0
0x0044 <+68>: andeq r0, r0, r0
```



**Huh??**

**So...**

**Let's avoid compiler  
optimisations!**

- All program variables are moved onto the stack before anything else
- Register spilling (> -O0): the register value is moved to the stack  
⇒ **Information leakage!**
- Bigger code size -> larger attack surface  
⇒ **More potential vulnerabilities**

Dump of assembler code for function subBytes:

```
0x84e4 <+0>: push {r11}          ; (str r11, [sp, #-4]
0x84e8 <+4>: add r11, sp, #0
0x84ec <+8>: sub sp, sp, #12
0x84f0 <+12>: mov r3, #0
0x84f4 <+16>: str r3, [r11, #-8]
0x84f8 <+20>: b 0x8530 <subBytes+76>
0x84fc <+24>: ldr r2, [pc, #68] ; <subBytes+100>
0x8500 <+28>: ldr r3, [r11, #-8]
0x8504 <+32>: add r3, r2, r3
0x8508 <+36>: ldrb r3, [r3]
0x850c <+40>: ldr r2, [pc, #56] ; <subBytes+104>
0x8510 <+44>: ldrb r2, [r2, r3]
0x8514 <+48>: ldr r1, [pc, #44] ; <subBytes+100>
0x8518 <+52>: ldr r3, [r11, #-8]
0x851c <+56>: add r3, r1, r3
0x8520 <+60>: strb r2, [r3]
0x8524 <+64>: ldr r3, [r11, #-8]
0x8528 <+68>: add r3, r3, #1
0x852c <+72>: str r3, [r11, #-8]
0x8530 <+76>: ldr r3, [r11, #-8]
0x8534 <+80>: cmp r3, #15
0x8538 <+84>: bls 0x84fc <subBytes+24>
0x853c <+88>: sub sp, r11, #0
0x8540 <+92>: pop {r11} ; (ldr r11, [sp], #4)
0x8544 <+96>: bx lr
0x8548 <+100>: andeq r0, r1, r4, lsl #15
0x854c <+104>: muleq r1, r4, r7
```

# Side-Channel Attacks

ISSISP 2017 – Gif-sur-Yvette  
2017-07-21

Damien Couroussé, CEA – LIST / LIALP; Grenoble Université Alpes  
[damien.courousse@cea.fr](mailto:damien.courousse@cea.fr)



Centre de Grenoble  
17 rue des Martyrs  
38054 Grenoble Cedex



Centre de Saclay  
Nano-Innov PC 172  
91191 Gif sur Yvette Cedex

