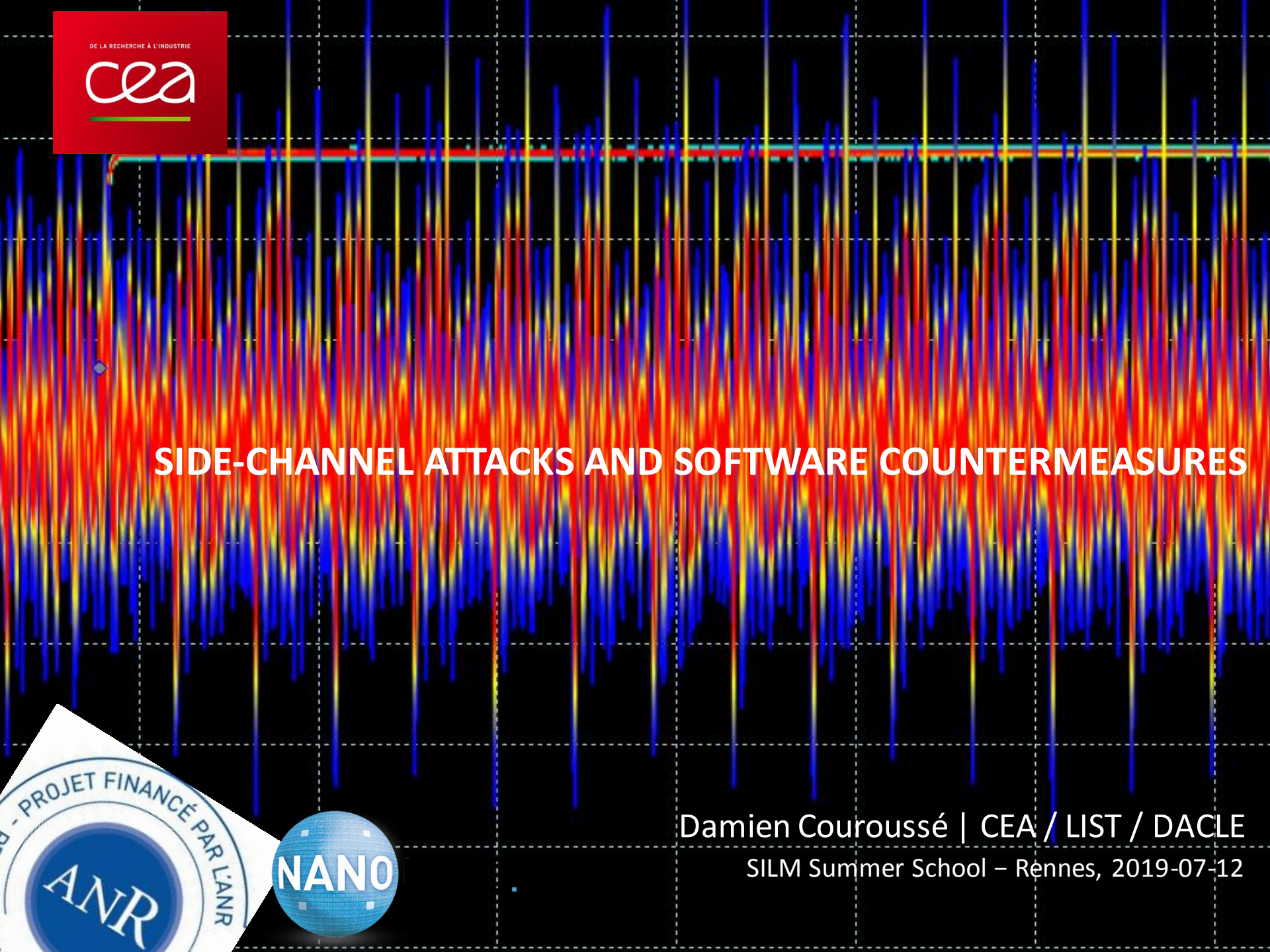


SIDE-CHANNEL ATTACKS AND SOFTWARE COUNTERMEASURES

Damien Couroussé | CEA / LIST / DACLE

SILM Summer School – Rennes, 2019-07-12

CEA TECH DACLE

leti Grenoble

list Saclay

DACLE
Architectures, IC Design & Embedded Software Division

300 members
160 permanent researchers

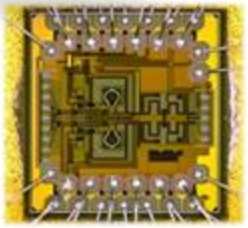
60 PhD students & postdocs

> 150 scientific papers per year

45 patents per year

Digital design

Programming



Analog & MEMs

Signal processing

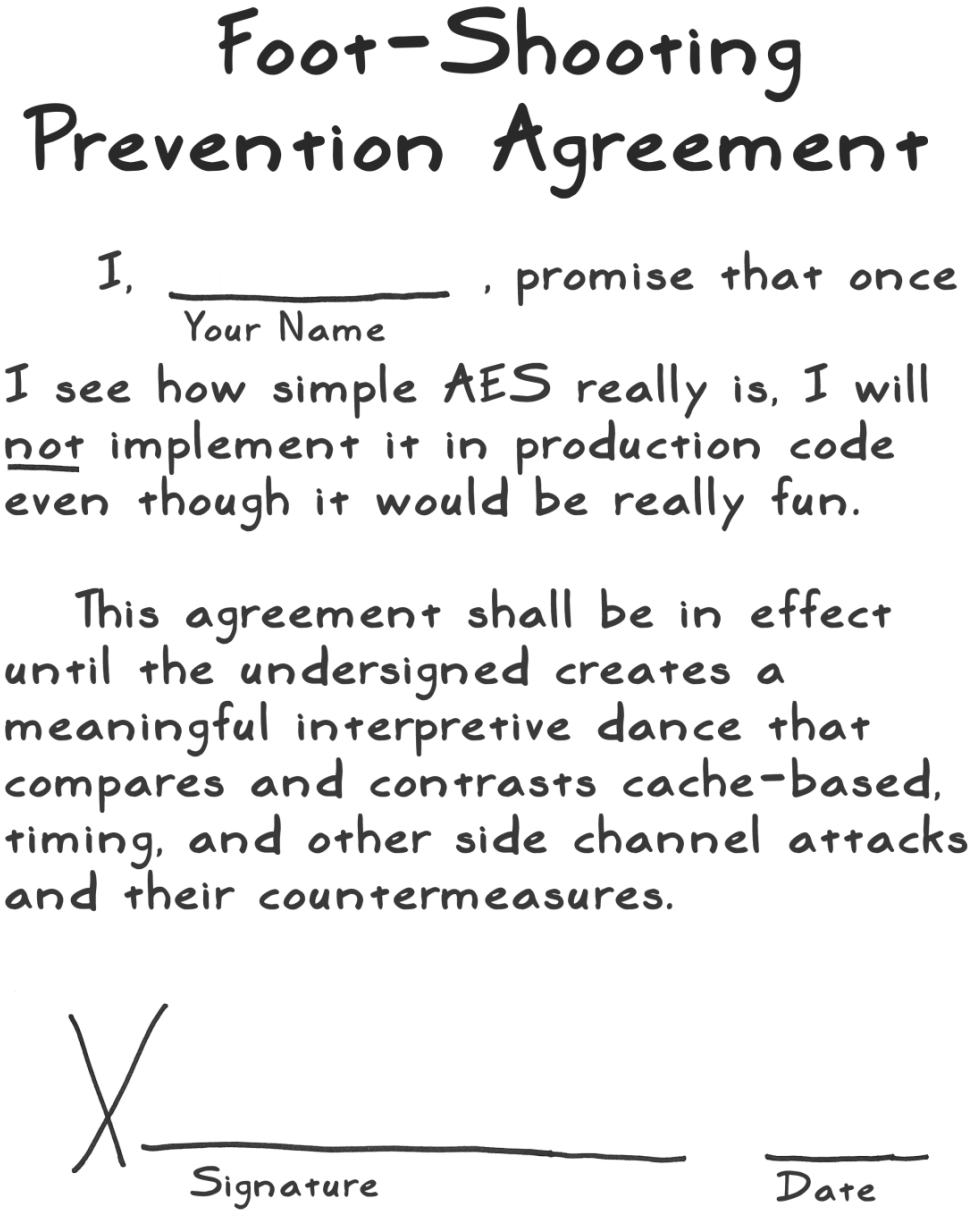
Imaging

Test

PhD and post-doc offers:

http://www-instn.cea.fr/formations/formation-par-la-recherche/doctorat/liste-des-sujets-de-these.html

http://www-instn.cea.fr/formations/formation-par-la-recherche/post-doctorat/liste-des-propositions-de-post-doctorat.html



PHYSICAL ATTACKS

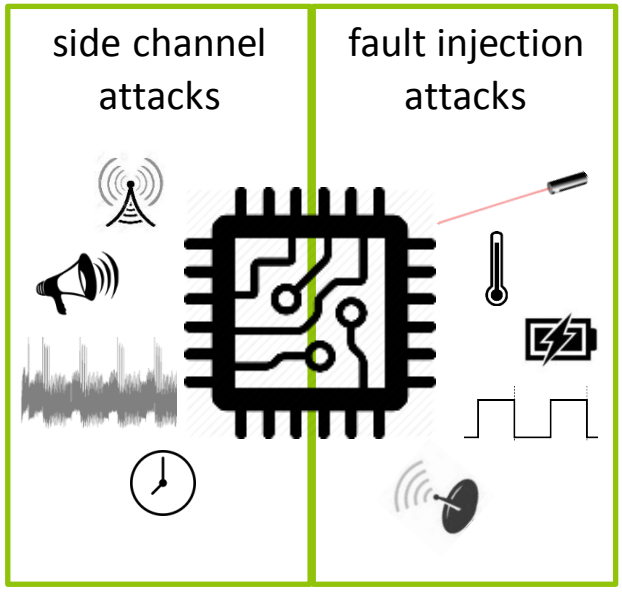
(semi-invasive)

A major threat against secure embedded systems

- The most effective attacks against implementations of cryptography
- Relevant against many parts of CPS/IoT: bootloaders, firmware upgrade, etc.
- Recently used to leverage software vulnerabilities [1]

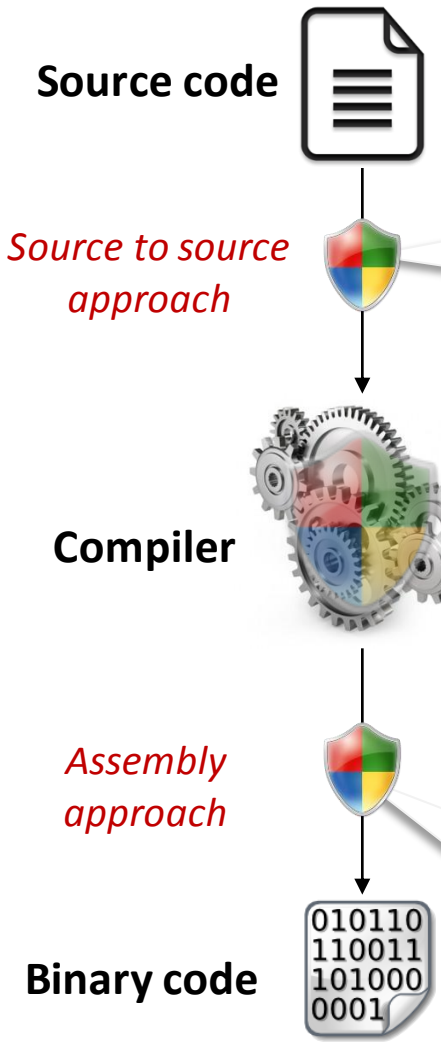
In practice,

- An attacker mostly uses logical attacks if the target is unprotected (e.g. typical IoT devices): buffer overflows, ROP, protocol vulnerabilities, etc.
- All high security products embed countermeasures against side-channel and fault injection attacks. E.g. Smart Cards, payTV, military-grade devices.
  - Using a combination of hardware *and* software countermeasures
- Tools for Side-channel and fault injection are getting really affordable



[1] A. Cui and R. Housley, 'BADFET: Defeating Modern Secure Boot Using Second-Order Pulsed Electromagnetic Fault Injection', presented at the WOOT, 2017.

AUTOMATED APPLICATION OF COUNTERMEASURES WITH A COMPILER



- ✓ Access to program's semantics (e.g. secret variable)
- × Security properties are not guaranteed, post compilation
- × Corollary: can lead to bigger overheads

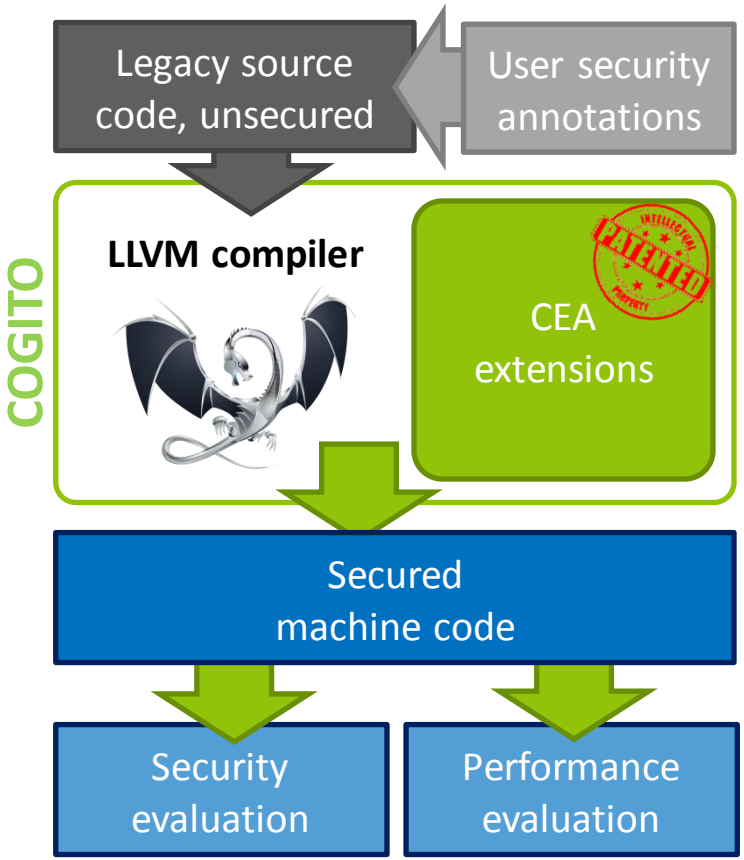
← **our approach**

- ✓ Access to program semantics
- ✓ Control over machine code
- ✓ Benefit from compiler optimisations
- × Implementation within the compiler is difficult
- ×/✓ Focus on generic countermeasures

- ✓ Naturally fits to low-level / machine code protection schemes
- × (Re-)construction of a program representation is difficult
- × Mostly *ad hoc* protection schemes

Automated application of software countermeasures against physical attacks

➔ A toolchain for the compilation of secured programs



Several countermeasures

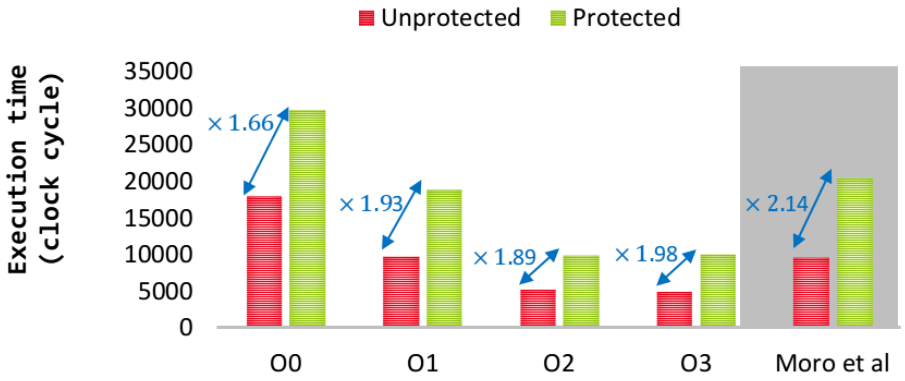
- **Fault tolerance**, including multiple fault injections
- **Execution Integrity & Control-Flow Integrity**
  - Detection of perturbations on the instruction path, at the granularity of a single machine instruction
- **Side channel hiding**

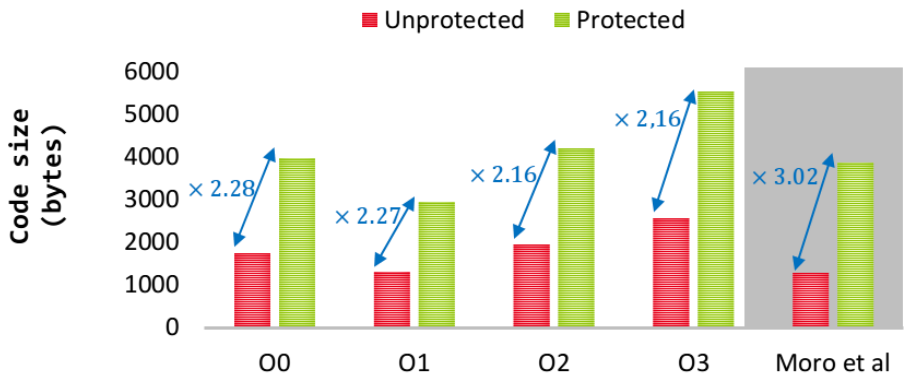
Tools for **security and performance evaluations**

Based on **LLVM**: an industry-grade, state-of-the art compiler (competitive with GCC)

Objective: the program is not perturbed by the injection of faults

- Countermeasure based on a protection scheme **formally verified for the ARM** architecture, against instruction skips [Moro et al., 2014, Barry et al. 2016]
- **Automatic application** by the compiler
- Allow to **parameterize** level of protection
- **Generalisation** of [Moro et al., 2014] to **multiple faults** of **configurable width**
- Target: ARM Cortex-M cores



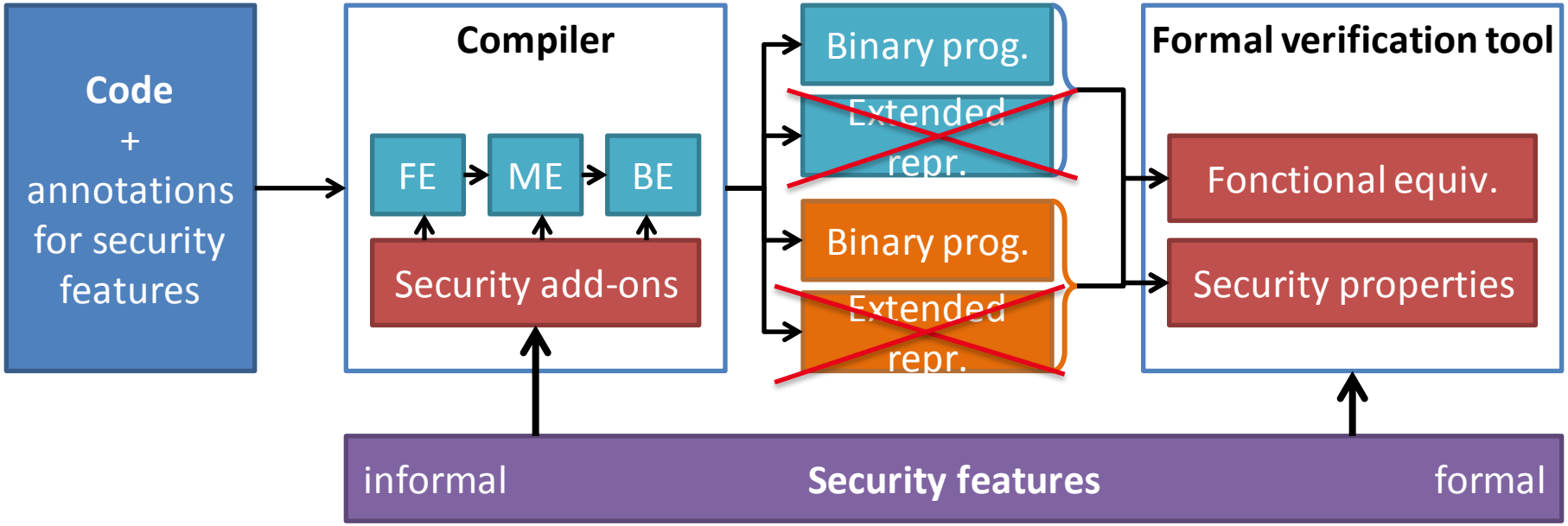


- **Fine-grained countermeasure** applied to critical functions **reduces the execution overhead** below x1.23 and size overheads below x1.12 [Barrys' thesis, 2017]

[Moro et al., 2014] Moro, N., Heydemann, K., Encrenaz, E., & Robisson, B. (2014). Formal verification of a software countermeasure against instruction skip attacks. Journal of Cryptographic Engineering, 4(3), 145-156.
[Barry et al. 2016] Barry, T., Couroussé, D., & Robisson, B. (2016, January). Compilation of a Countermeasure Against Instruction-Skip Fault Attacks. In Proceedings of the Third Workshop on Cryptography and Security in Computing Systems (pp. 1-6). ACM.

SECURING AND VERIFYING PROGRAMS

- **Compilation:** automation of the application of software countermeasures against fault attacks and side-channel attacks
- **Functional verification:** of the secured machine code (equivalence with an unprotected version of the same program)
- **Security verification:** correctness of the applied countermeasures w.r.t a security model



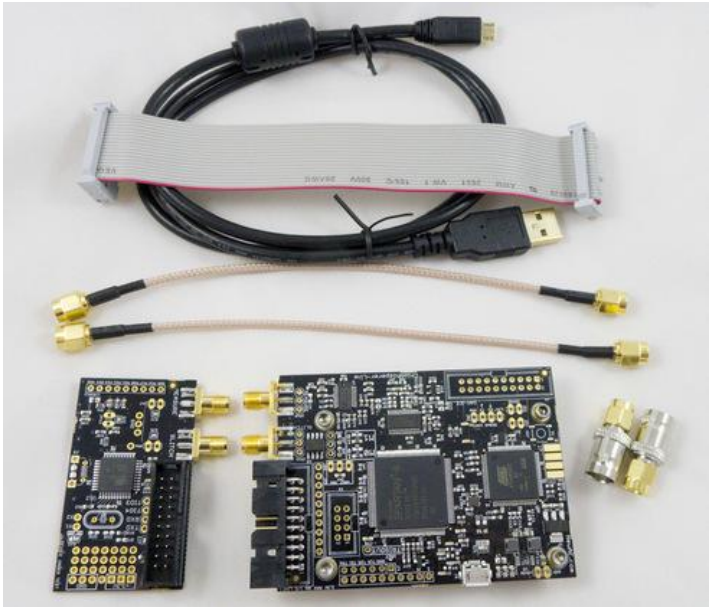
On-going joint work with LIP6, Paris (PROSECCO – ANR 2015)

SIDE-CHANNEL ATTACKS

« PHYSICAL ATTACKS IS SCI-FI »

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

- Require dedicated HW attack benches, can be quite expensive, especially for fault injection (e.g., laser benches)
- We also find low cost ones
  - E.g. *The ChipWhisperer*, starting at ~ 300€
- Require human expertise, but not more than other attacks



https://newae.com/tools/chipwhisperer

« PHYSICAL ATTACKS IS SCI-FI »

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"

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,adi.shamir}@weizmann.ac.il
†Dalhousie University, Halifax, Canada
coflynn@dal.ca

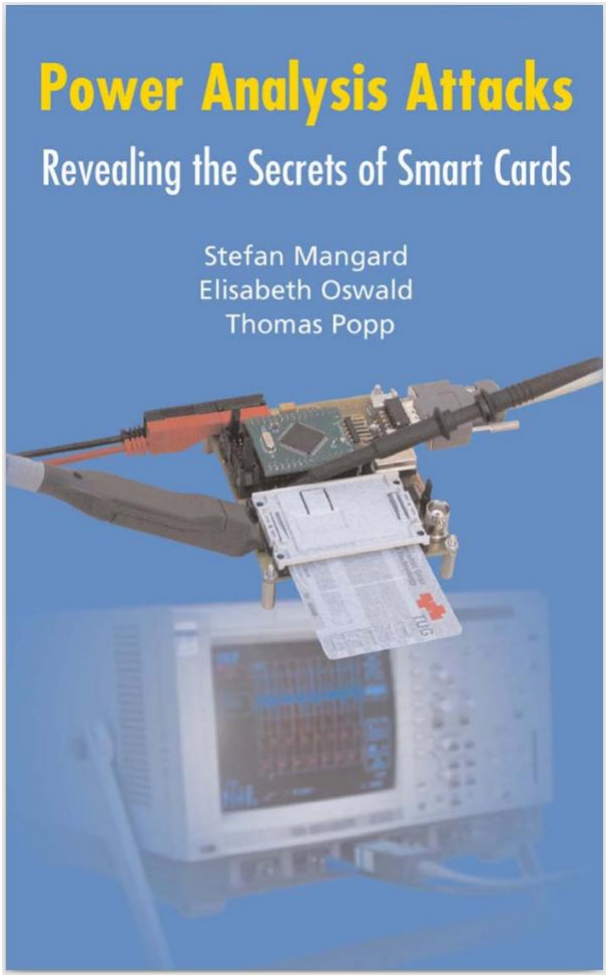
- **Philips Hue Smart lamp**
  - ZigBee protocol
- **Uploading malicious firmware with OTA update**
  - Discovered the hex command code for OTA update
  - Firmware is protected with a single global key! Using symmetric crypto (AES-CCM).
- **Attack path**
  - Get access to the key → **side-channel attack 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.

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

THE “DPA” BOOK

The most comprehensive book about side-channel attacks

- Excellent introduction to side-channel attacks
- Published in 2007: does not cover recent attacks and countermeasures, but still really useful

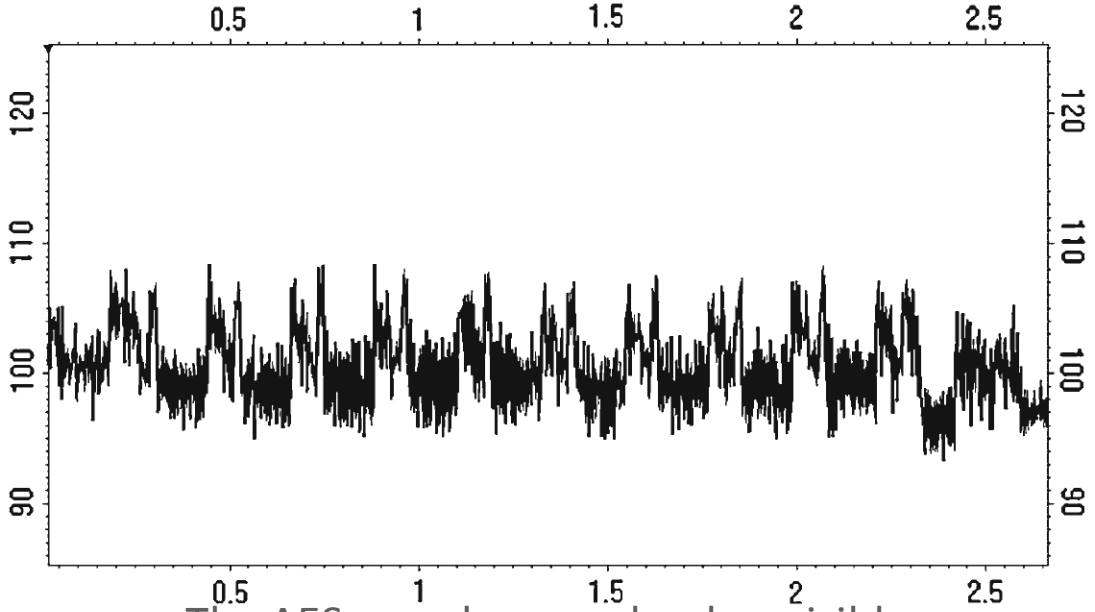


S. Mangard, E. Oswald, and T. Popp, Power analysis attacks: Revealing the secrets of smart cards, vol. 31. Springer, 2007.

SIMPLE POWER ANALYSIS (SPA)

Direct interpretation of power consumption measurements
Extraction of information by inspection of single side-channel traces

SPA on AES [Kocher, 2011]



The AES rounds are « clearly » visible

- 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.

SIMPLE POWER ANALYSIS (SPA)

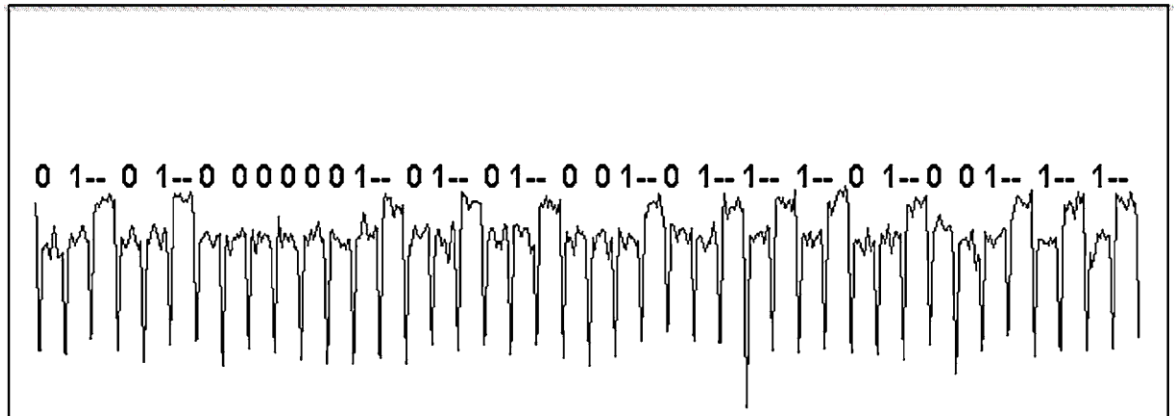
SPA on RSA [Kocher, 2011]

```
-- Computing c = b ^ e mod 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 [DPA book]:

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

$$P_{total} = P_{exploitable} + P_{switching.noise} + P_{electronic.noise} + P_{const}$$ [DPA book]

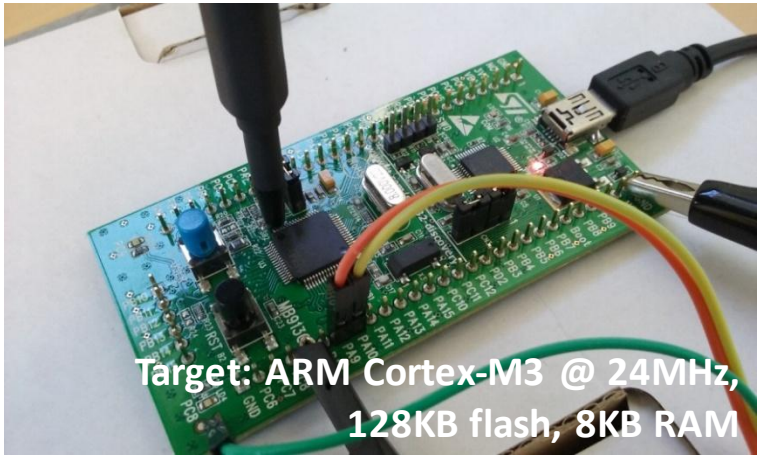
needle haystack

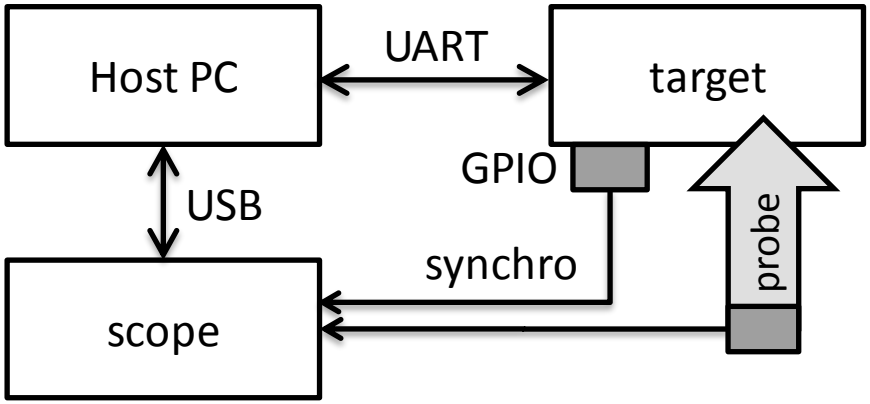
- Power signal: a static and a dynamic component.
  - Static component: power consumption of the gate states → a * HW(state)
  - Dynamic component: power consumption of transitions in gate states → b * HD(state[i], state[i-1])
- Other needles & stacks
  - Electromagnetic emissions
  - Execution time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

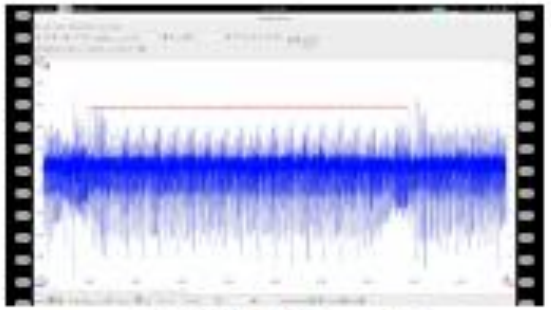
CPA – MEASUREMENT SETUP

- The AES key is fixed
- A GPIO trigger is used to facilitate the trace measurements
- The attacker
  - either knows public data: plaintexts if decrypting, ciphertexts if encrypting
  - or controls the encryption/decryption function
- Text chosen attack:
  - Generate D random plaintexts
  - Ask the cipher text to the target
  - Record the EM trace during encryption
- **Do the computation analysis!**

Target: STM32 – ARM Cortex-M3 @ 24MHz, 128KB flash, 8KB RAM

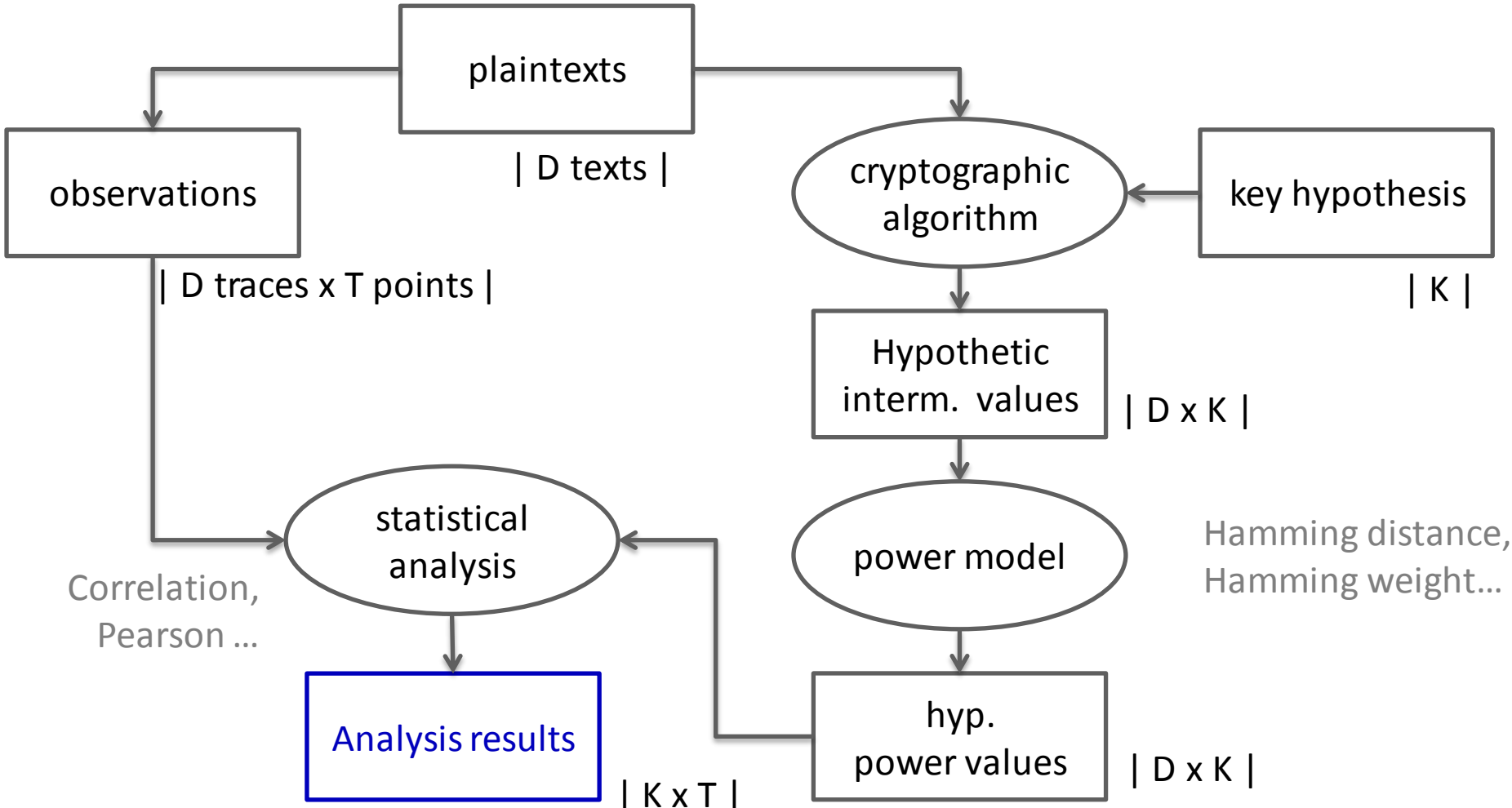




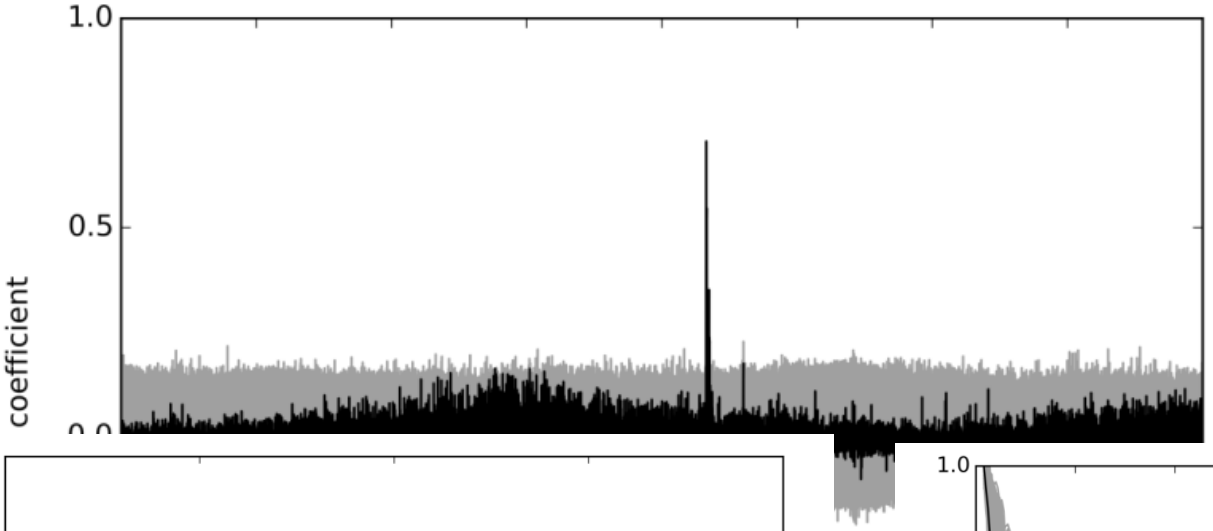

COGITO.mp4

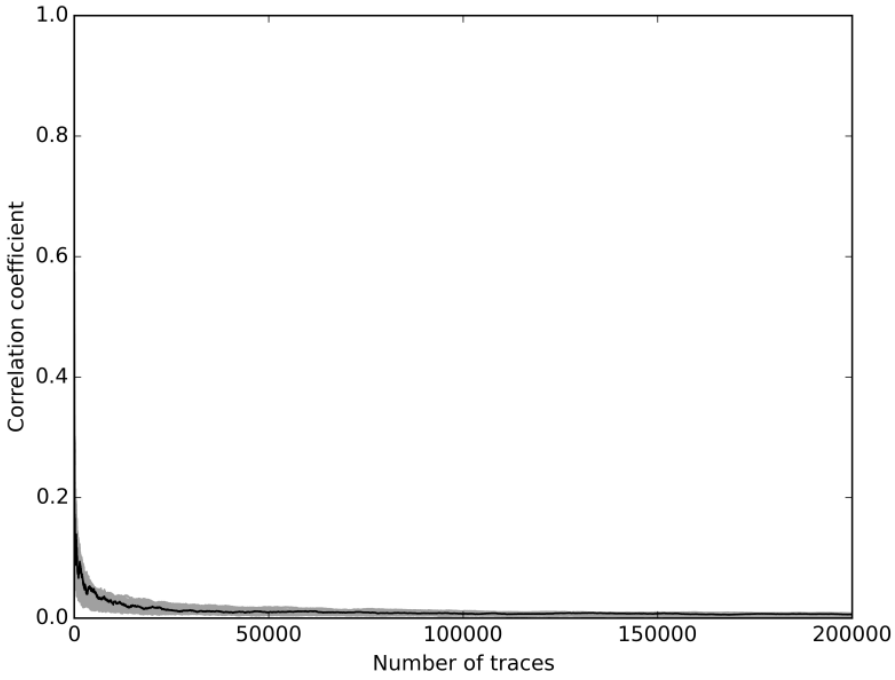
CPA – ANALYSIS

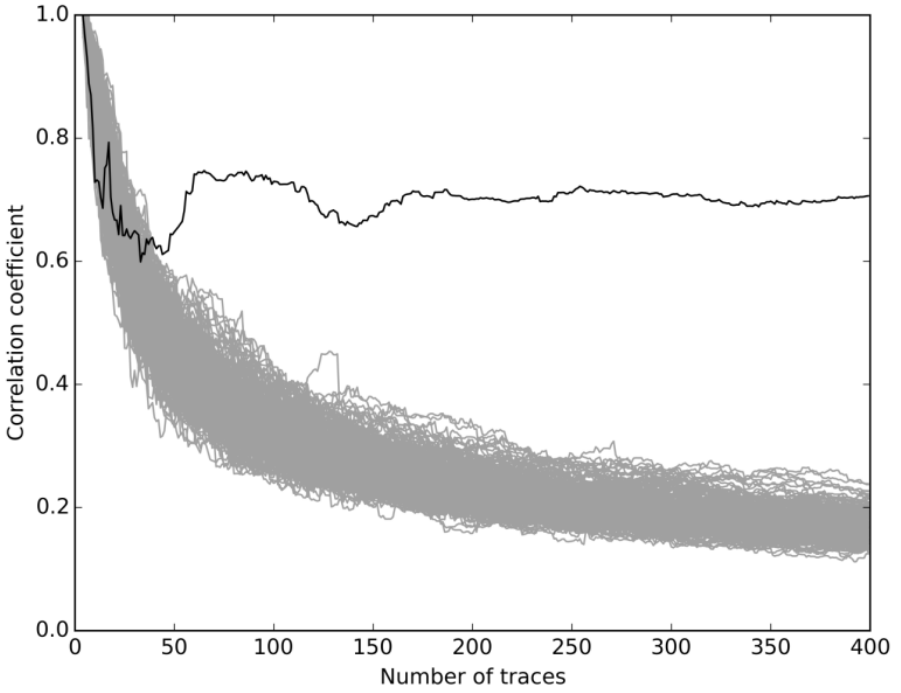
m: plaintext -> controlled by the attacker or observable
k: cipher key -> unknown to the attacker



CPA RESULTS







ESTIMATING THE SUCCESS OF AN ATTACK

Success rate: success probability of a successful attack

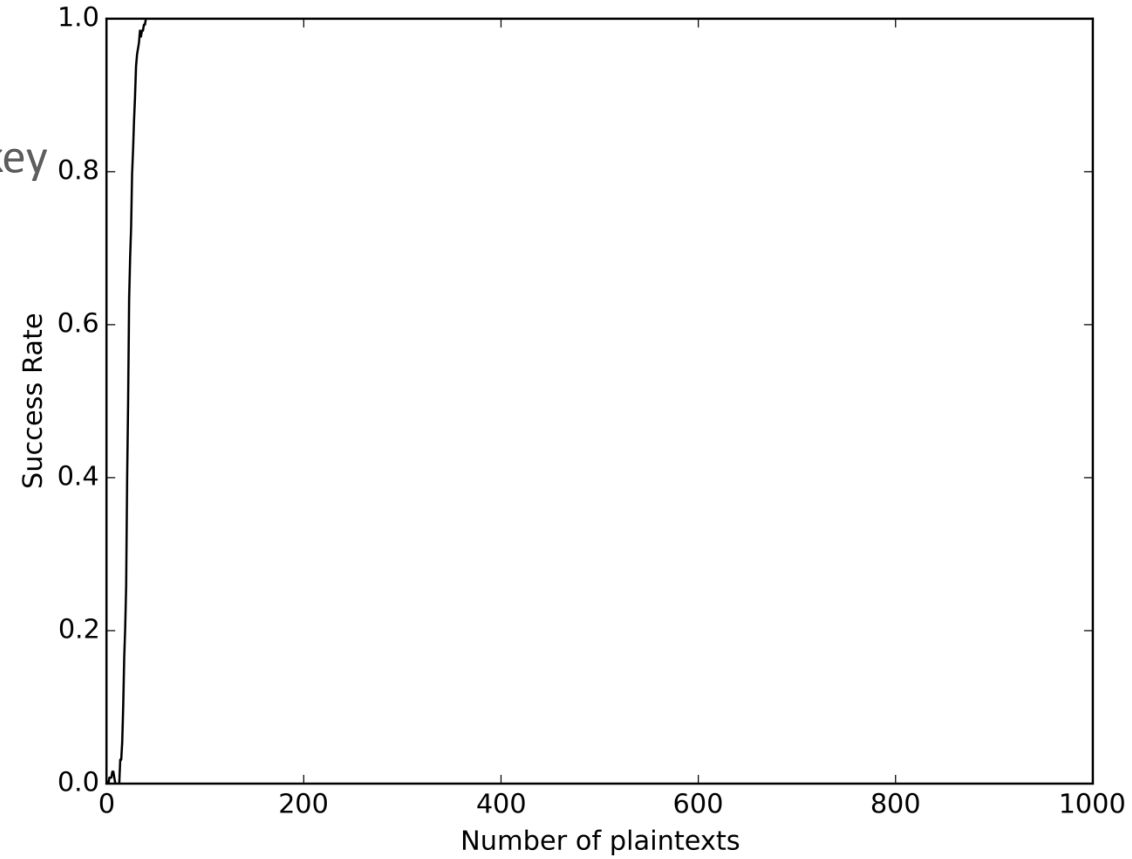
$$SR = Pr[\ A(E_{k0}, L) = k_0\ ]$$

- A — side-channel attack
- k_0 — correct key
- E_{k0} — encryption with correct key
- L — leakage

n-order success rate?

PGE. Partial guessing entropy

Empirical evaluation at first order



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.

SECURITY EVALUATION?

- **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 the10th 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

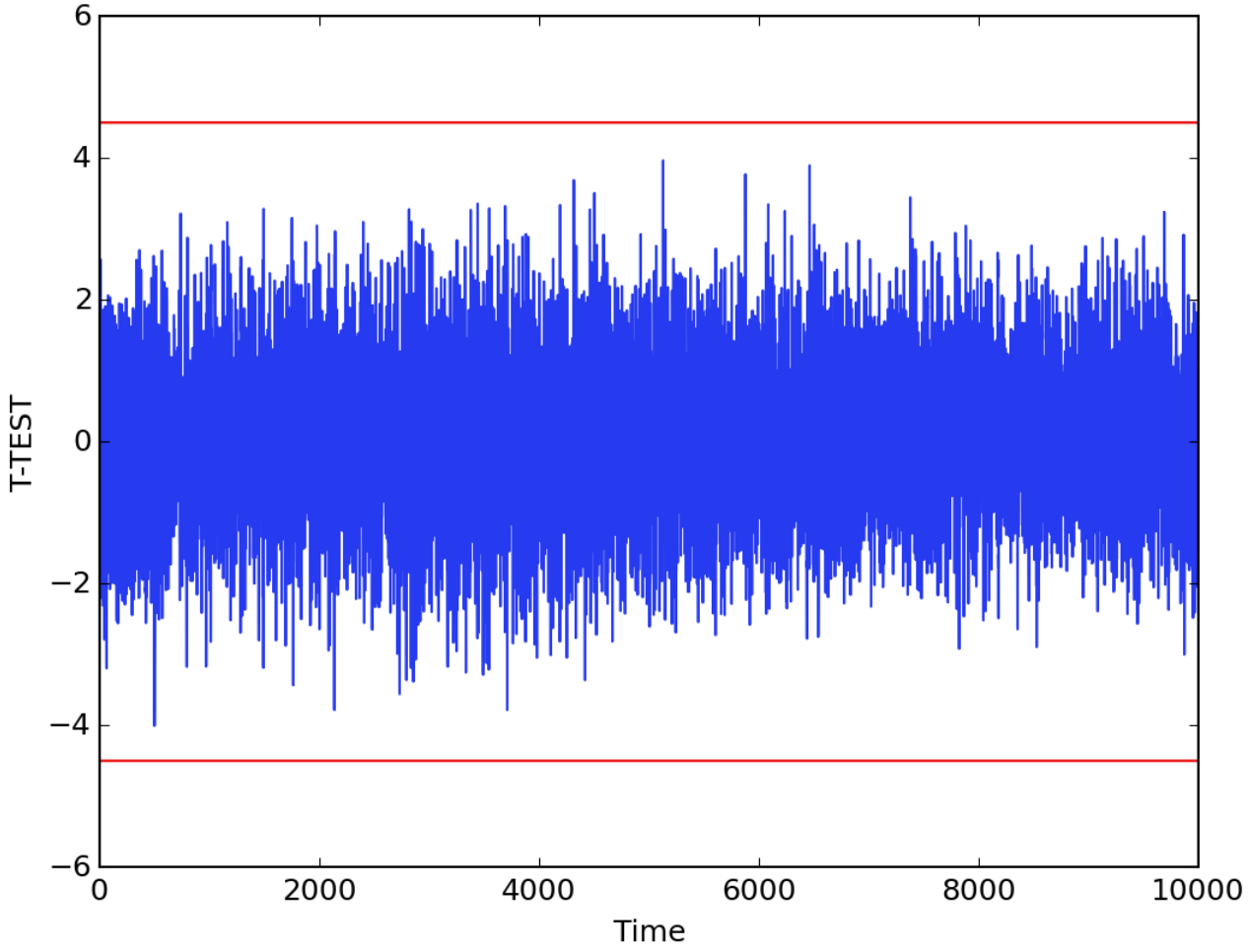
$$t = \frac{\mu_0 - \mu_1}{\sqrt{\frac{{s_0}^2}{n_0} + \frac{{s_1}^2}{n_1}}},$$

t > 4.5 → 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.

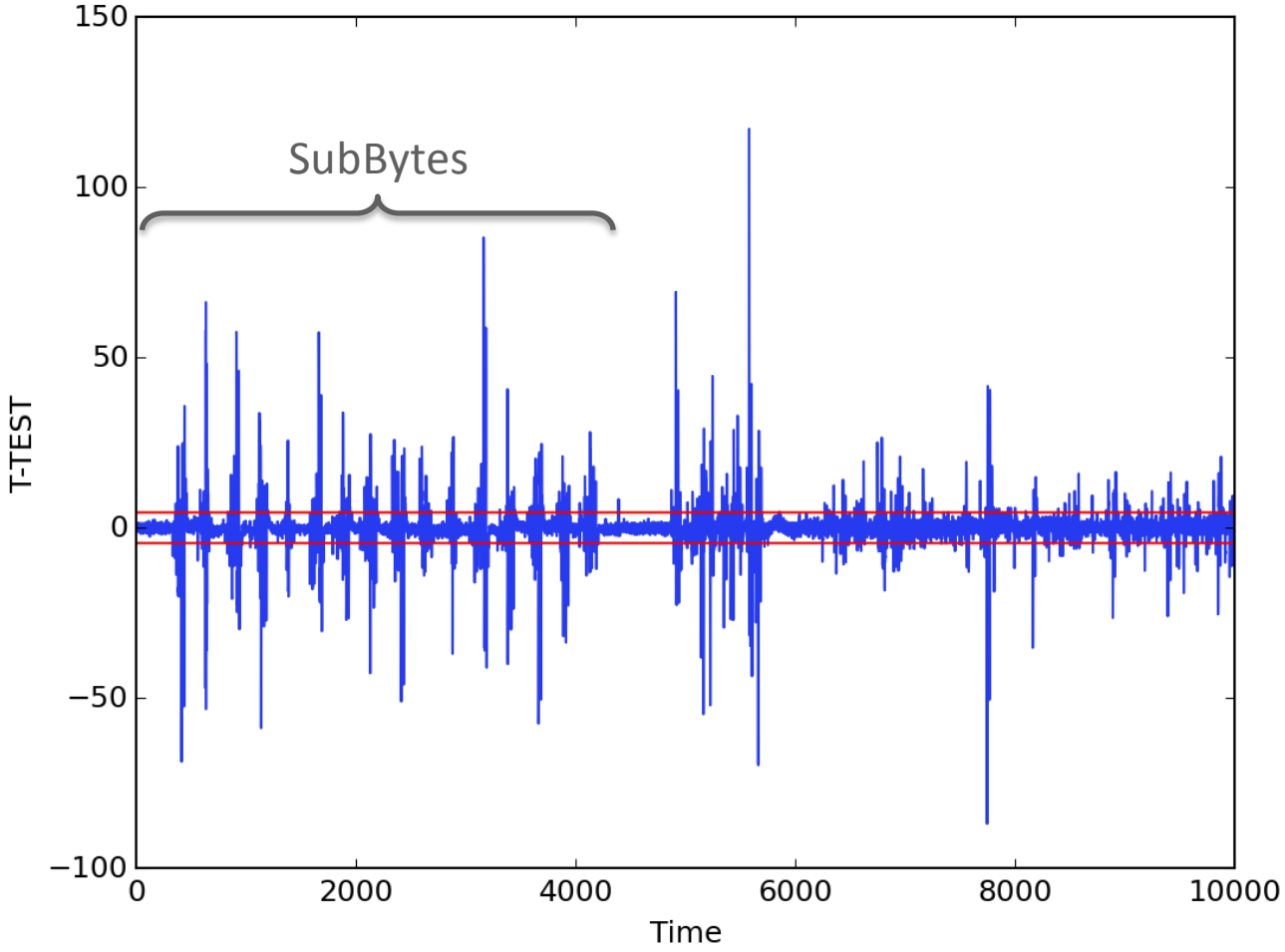
T. Schneider and A. Moradi, "Leakage Assessment Methodology - a clear roadmap for side-channel evaluations," 207, 2015.

T-TEST



NON-SPECIFIC T-TEST

Q0: fixed input plaintext
Q1: random input plaintext



SPECIFIC T-TEST

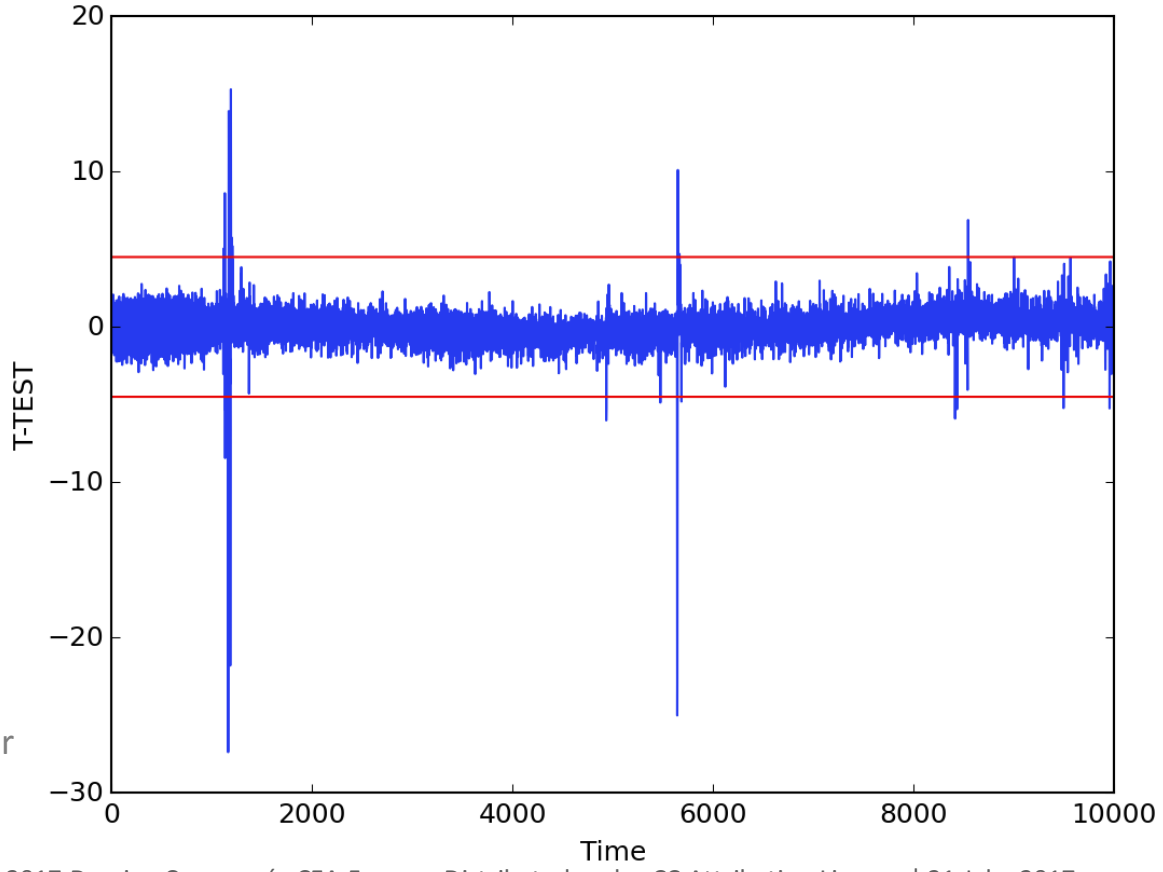
Mono-bit t-test:

$$\mathcal{Q}_0 = \{T_i \,|\, \text{target bit}(D_i) = 0\}, \qquad \mathcal{Q}_1 = \{T_i \,|\, \text{target bit}(D_i) = 1\}.$$

Bytewise t-test:

$$\mathcal{Q}_0 = \{T_i \,|\, \text{target byte}(D_i) = \text{x}\}, \qquad \mathcal{Q}_1 = \{T_i \,|\, \text{target byte}(D_i) \neq \text{x}\}.$$

Number of measurements for a security evaluation with a specific t-test?



T. Schneider and A. Moradi, 'Leakage Assessment Methodology - a clear roadmap for side-channel evaluations', 207, 2015.

DEFEATING THE T-TEST

- **Assuming two populations of N traces P_i and Q_j,**
  - N sufficiently large w.r.t. t-test criteria
  - Leakage exhibited on P, Q by a specific or non-specific t-test
  - Also assuming that information leakage is not spread over several consecutive samples
- **Create two new populations of traces such that:**
  - $P'_0 = P_0$
  - $P'_i[0; i-1] \leftarrow 0, \quad i \neq 0$
  - $P'_i[i; \ldots] \leftarrow P_i[0; \ldots], \quad i \neq 0$
  - Similar construction for Q

Signal-to-noise ratio

$$P_{total} = P_{operation} + P_{data} + P_{noise}$$

$$P_{tot} = P_{exploi} + P_{sw.noi} + P_{el.noi} + K$$

$$SNR = \frac{Var(sign)}{Var(noise)} = \frac{Var(P_{exploi})}{Var(P_{sw.noi} + P_{el.noi})}$$ [DPA book]

- Powerful metric, less sensitive to the modus operandi
- Still depends on the attacker's model

NICV (normalised inter-class variance)

$$SNR = \frac{Var(E(L|X))}{Var(L)} = \frac{1}{\frac{1}{SNR}+1}$$ X: known plaintext or ciphertext data; L: observations

S. Bhasin, J.-L. Danger, S. Guilley, and Z. Najm, 'NICV: Normalized Inter-Class Variance for Detection of Side-Channel Leakage', 717, 2013.

COUNTER-MEASURES AGAINST SIDE-CHANNEL ATTACKS

MASKING & HIDING

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.

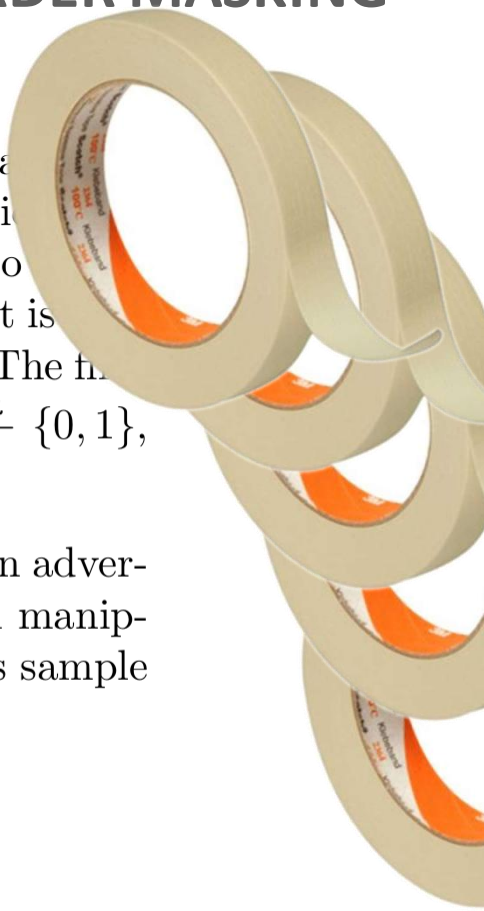
[DPA book]

- **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.**

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

HIGHER-ORDER MASKING



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

Denoting the vector of shares $(x_1, x_2, \ldots, x_m)$ 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:

$$\mathsf{L}_1(\bar{x}) = \left(\sum_{i=1}^{m} \alpha_i \cdot x_i \right) + N,$$

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

HIDING

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**.

HIDING

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 -> part of the observations containing useful data (« *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.

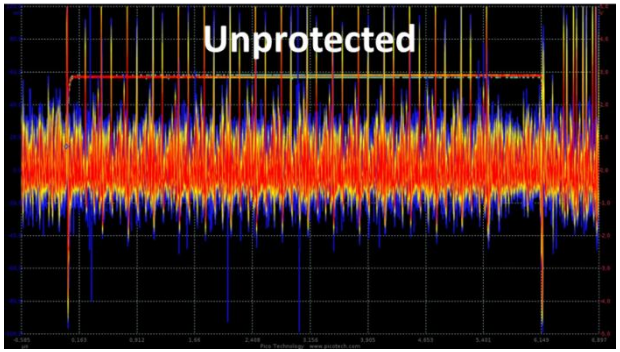
CODE POLYMORPHISM

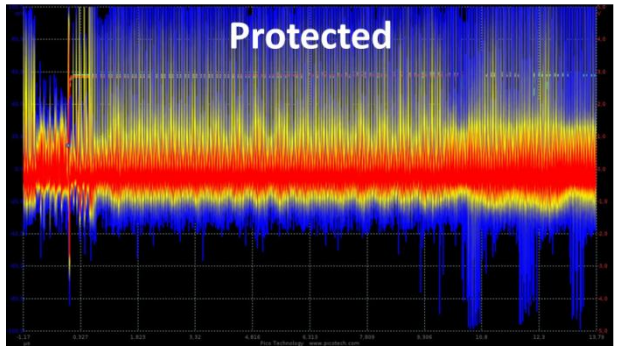
SIDE CHANNEL HIDING WITH CODE POLYMORPHISM

Code polymorphism: regularly changing the observable behavior of a program, at runtime, while maintaining unchanged its functional properties,

- **Protection against physical attacks: side channel & fault attacks**
  - Changes the spatial and temporal properties of the secured code
  - Can be combined with other state-of-the-Art HW & SW Countermeasures
- **Can run on low-end embedded systems with only a few kB of memory**
  - Illustrated below: STM32F1 microcontroler with 8kB of RAM

Compliant with certification standards (Common Criteria, CSPS, etc.)

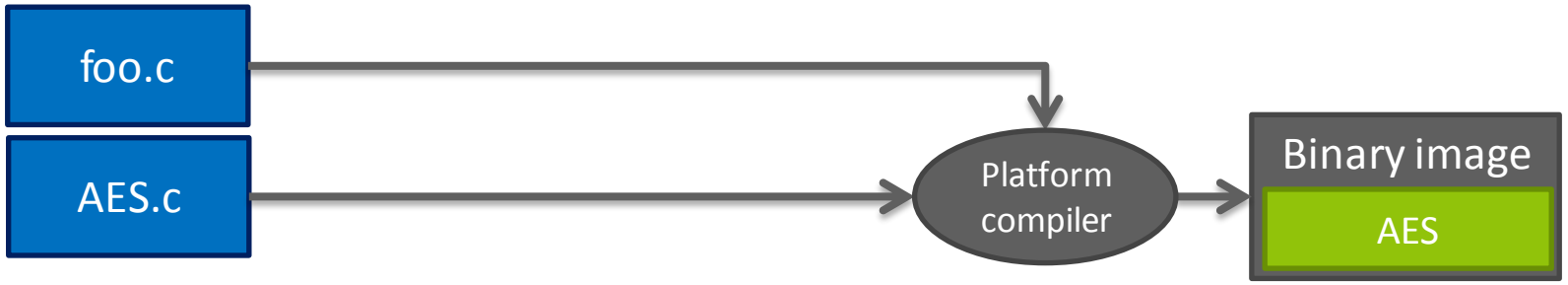




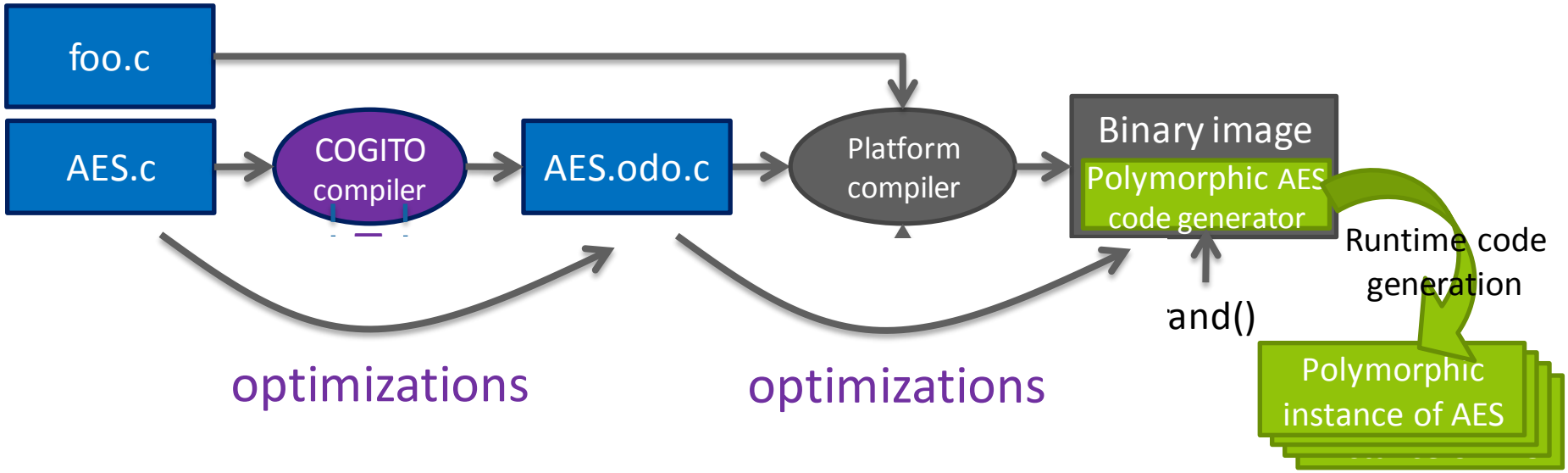
CODE POLYMORPHISM: WORKING PRINCIPLE

Runtime code generation for embedded systems

Reference version:



Polymorphic version, with COGITO:

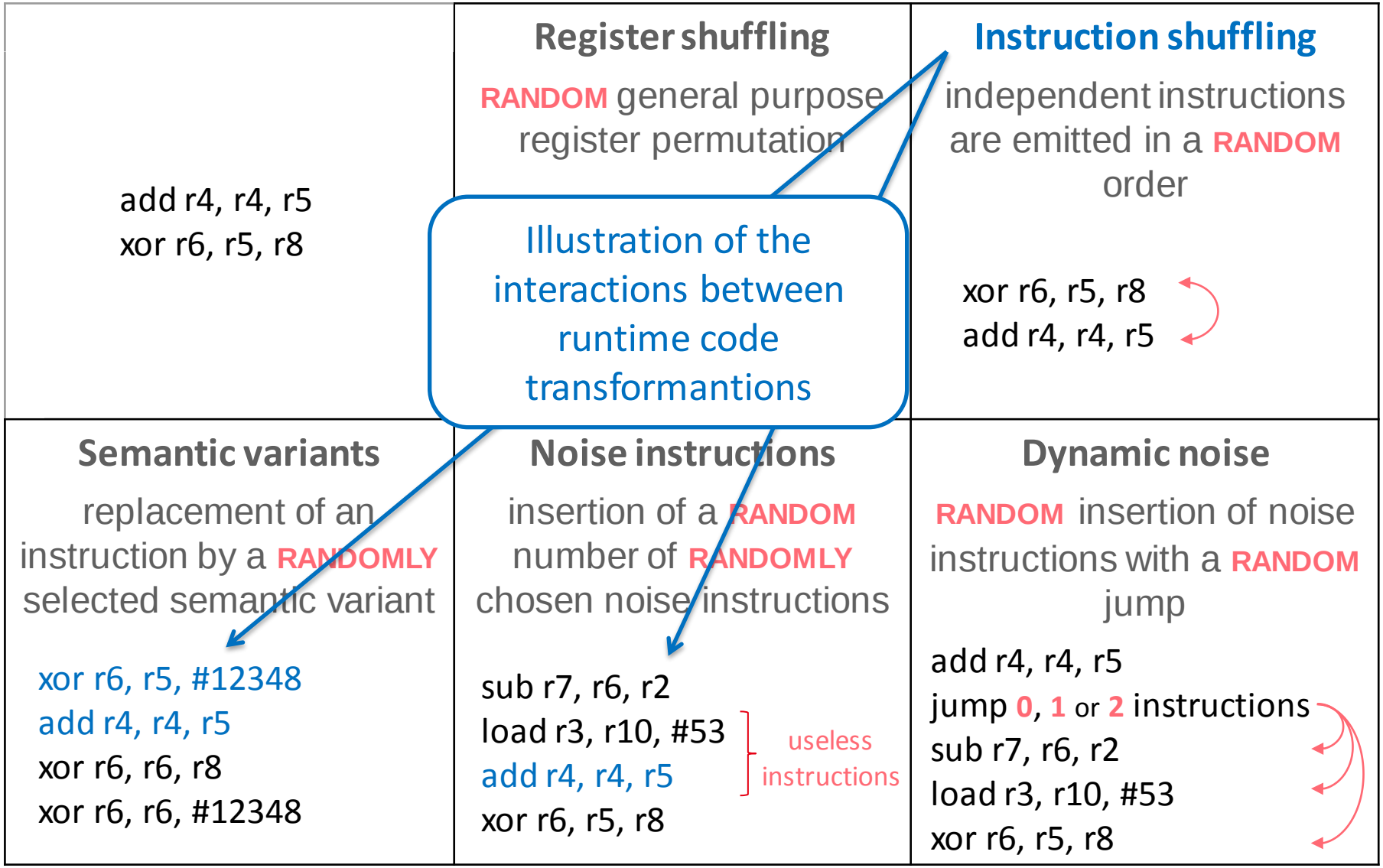


CODE TRANSFORMATIONS USED AT RUNTIME

add r4, r4, r5 xor r6, r5, r8	**Register shuffling** RANDOM general purpose register permutation r4 → r11 … add r11, r11, r7 xor r8, r7, r5	**Instruction shuffling** independent instructions are emitted in a RANDOM order xor r6, r5, r8 add r4, r4, r5

Semantic variants
replacement of an instruction by a RANDOMLY selected semantic variant
add r4, r4, r5
xor r6, r5, #12348
xor r6, r6, r8
xor r6, r6, #12348
	Noise instructions
	insertion of a RANDOM number of RANDOMLY chosen noise instructions
	add r4, r4, r5
	sub r7, r6, r2
	load r3, r10, #53
	xor r6, r5, r8
	useless instructions
		Dynamic noise
		RANDOM insertion of noise instructions with a RANDOM jump
		add r4, r4, r5
		jump 0, 1 or 2 instructions
		sub r7, r6, r2
		load r3, r10, #53
		xor r6, r5, r8

CODE TRANSFORMATIONS USED AT RUNTIME



TOOLCHAIN CONFIGURABILITY

Period of regeneration	**Register shuffling**	**Instruction shuffling**
$\mathbb{N}$ (+ custom regeneration policies)	$\{0, 1\}$	$\{0, 1\}$

Total configuration space:
$\{0, 1\}^2 \times \{0, 1, 2\}^2 \times \mathbb{R} \times \mathbb{N}^3$

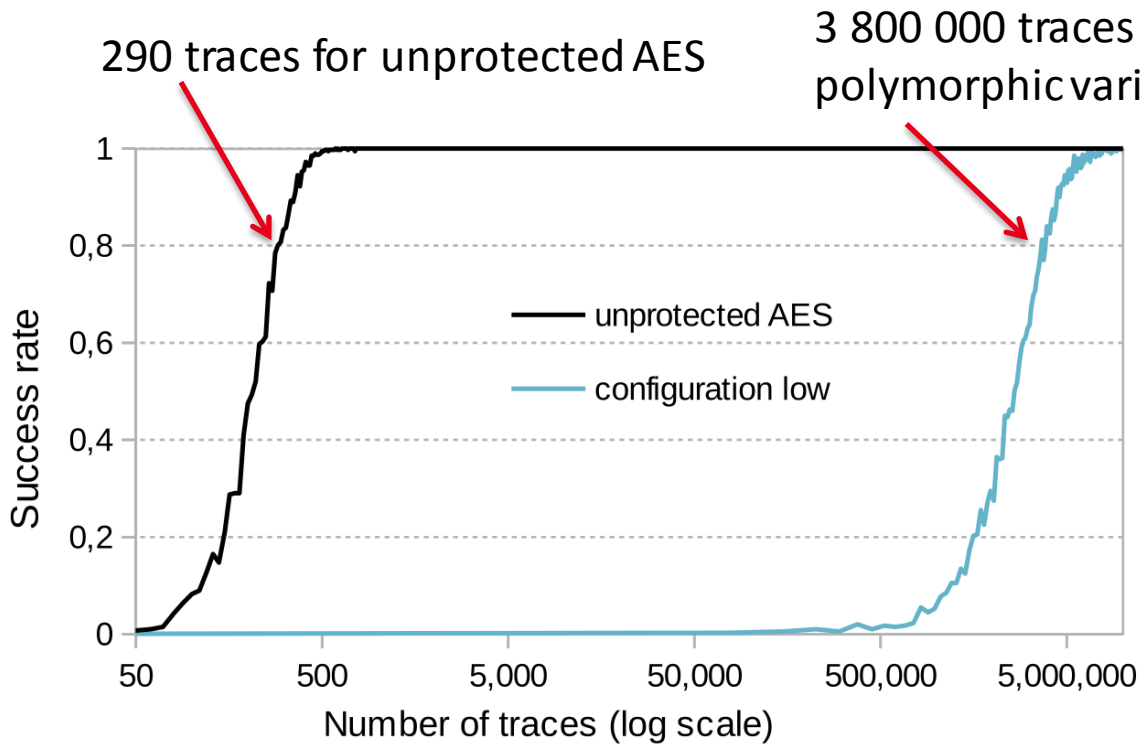
Semantic variants	**Noise instructions**	**Dynamic noise**
$\{0, 1, 2\}$ (+ custom extensions)	$\{0, 1, 2\} \times \mathbb{R} \times \mathbb{N}$ (+ custom probabilistic models)	$\mathbb{N}$ (+ custom noise sequences)

A huge number of polymorphic variants:

- 10 original machine instructions → 6.10^{42} variants
- **AES with 278 machine instructions** → 10^{27} **variants** (pessimist bound)

A ~~SECURITY EVALUATION~~

- Basis: polymorphic configuration with low variability
- Acquisition of traces from Electro-Magnetic observations
- CPA on SBOX 1st output with HW model
- Experimental platform not designed for security applications (hence the weak results on the unprotected version)



Experimental results

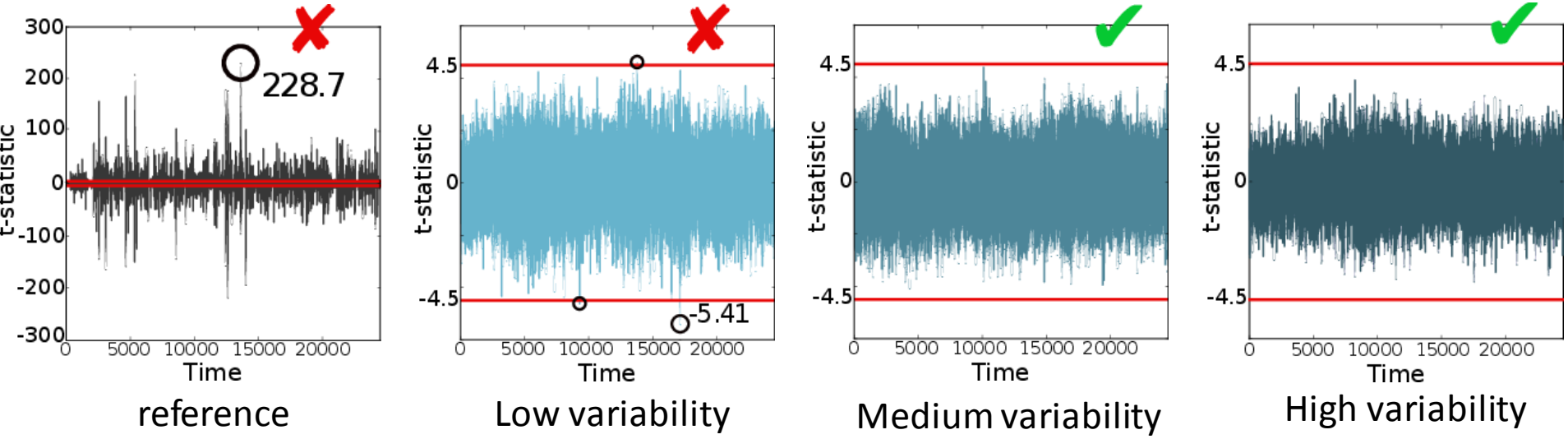
- This polymorphic version requires 13000x more traces
- Execution time overhead: x2.5 including generation cost

N. Belleville, D. Couroussé, K. Heydemann, and H.-P. Charles "Automated Software Protection for the Masses Against Side-Channel Attacks," *ACM TACO*, vol. 15, no. 4, pp. 47:1–47:27, Jan. 2019.

A ~~SECURITY EVALUATION~~

- **Polymorphism is a hiding countermeasure against side-channel attacks**
  - Does not remove information leakage; reduces SNR only
- **However, information leakage is sufficiently blurred such that it is not found in observation traces, with a confidence level of 99.999%**
- **Configurable level of polymorphism**

t-tests results



N. Belleville, D. Couroussé, K. Heydemann, and H.-P. Charles "Automated Software Protection for the Masses Against Side-Channel Attacks," *ACM TACO*, vol. 15, no. 4, pp. 47:1–47:27, Jan. 2019.

AUTOMATED APPLICATION OF CODE POLYMORPHISM

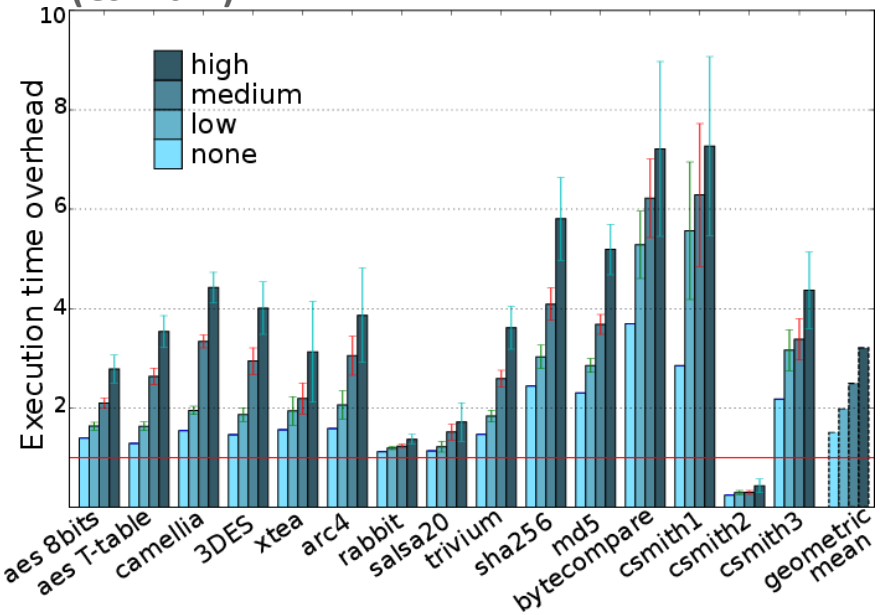
Declaration of polymorphism with a compiler option

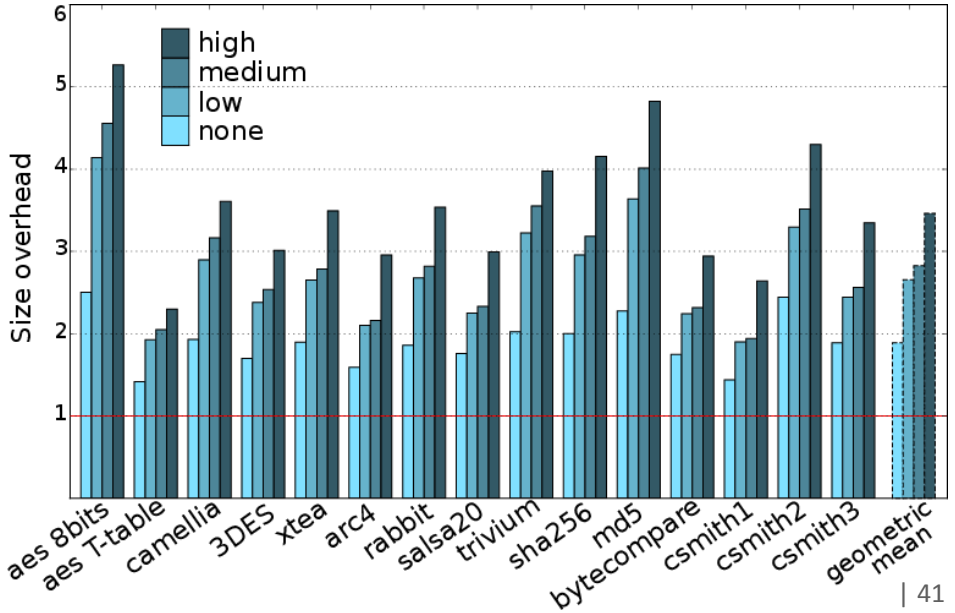
- -polymorphic-function foo will compile function foo into a polymorphic implementation,
- -polymorphic will compile all functions found in the compiled source le into polymorphic implementations.

Many configurable levels of polymorphic transformations => security/performance tradeoff

- Nature and parameters of the code transformations: random allocation of registers, semantic variants, instruction shuffling, insertion of noise instructions.
- Frequency and policy for runtime code regeneration
- Memory protections
- Leveraging OS-level features, e.g. concurrency

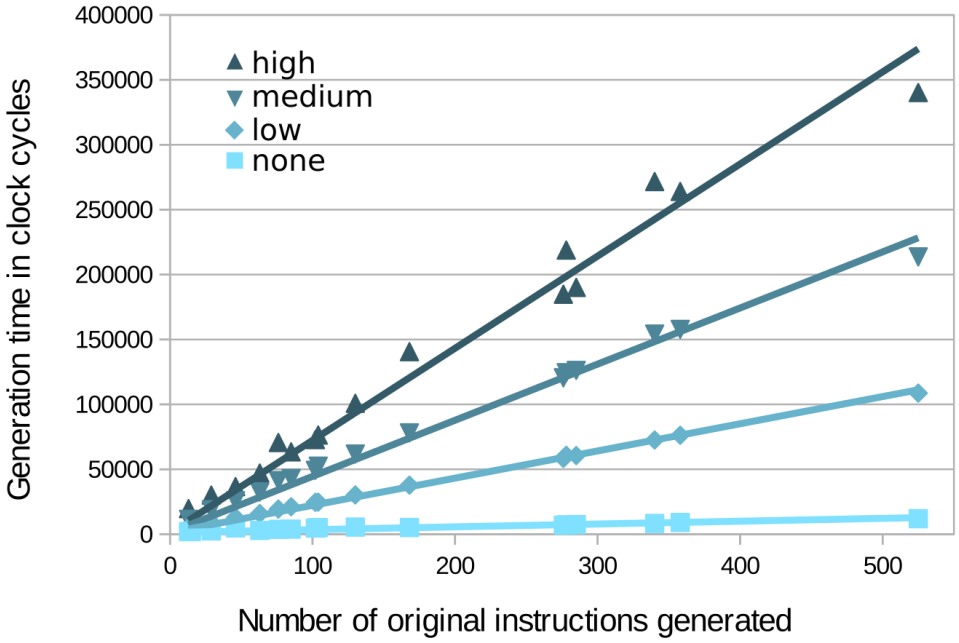
Components evaluated: ciphers, hash functions, simple authentication, random generated codes (Csmith*)





PERFORMANCE EVALUATION OF RUNTIME CODE GENERATION

Configuration	Execution time overhead (geo. mean)	Size overhead (geo. mean)
none ■	x1.40	x1.70
low ◆	x2.31	x2.87
medium ▼	x2.45	x3.44
high ▲	x4.03	x3.81



f'(x)=709 cycles per instruction

f'(x)=432 cycles per instruction

f'(x)=209 cycles per instruction

f'(x)=22 cycles per instruction

- Overheads depend on the selected configuration → trade-off w.r.t. the expected security level
- Runtime code generation achieved in linear complexity

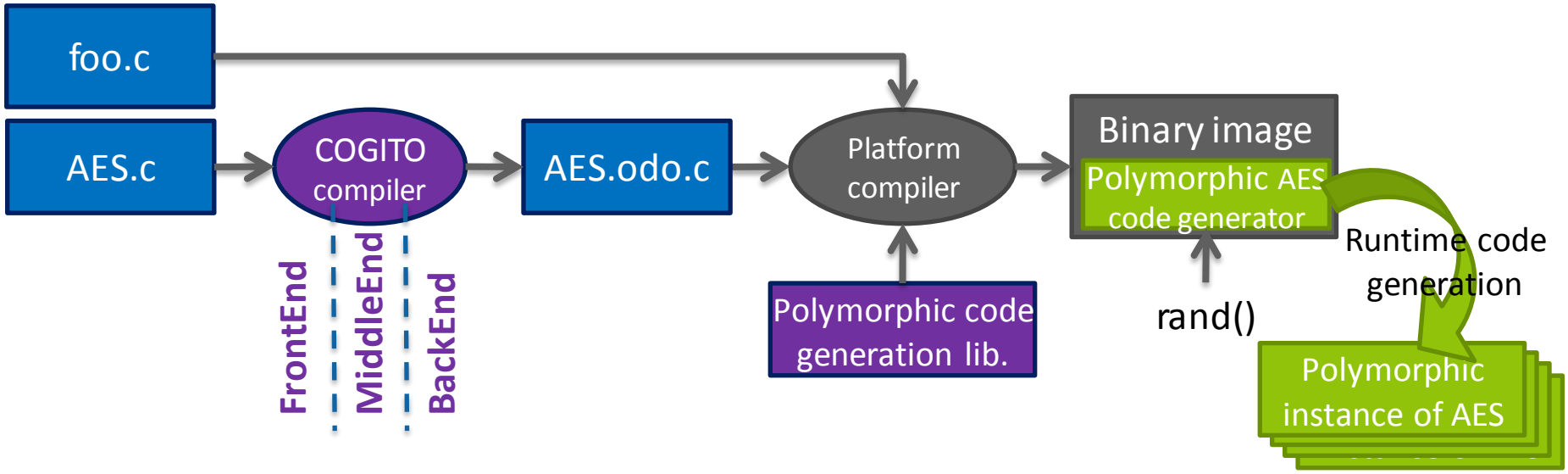
N. Belleville, D. Couroussé, K. Heydemann, and H.-P. Charles "Automated Software Protection for the Masses Against Side-Channel Attacks," *ACM TACO*, vol. 15, no. 4, pp. 47:1–47:27, Jan. 2019.

CODE POLYMORPHISM: CHALLENGES

Bottlenecks for the use of runtime code generation in embedded systems:

- Memory allocation of code buffers
  - No Operating System (no malloc), no virtual memory.
- Management of memory permissions (read, write, execute)
  - Runtime code generation requires write access to program memory

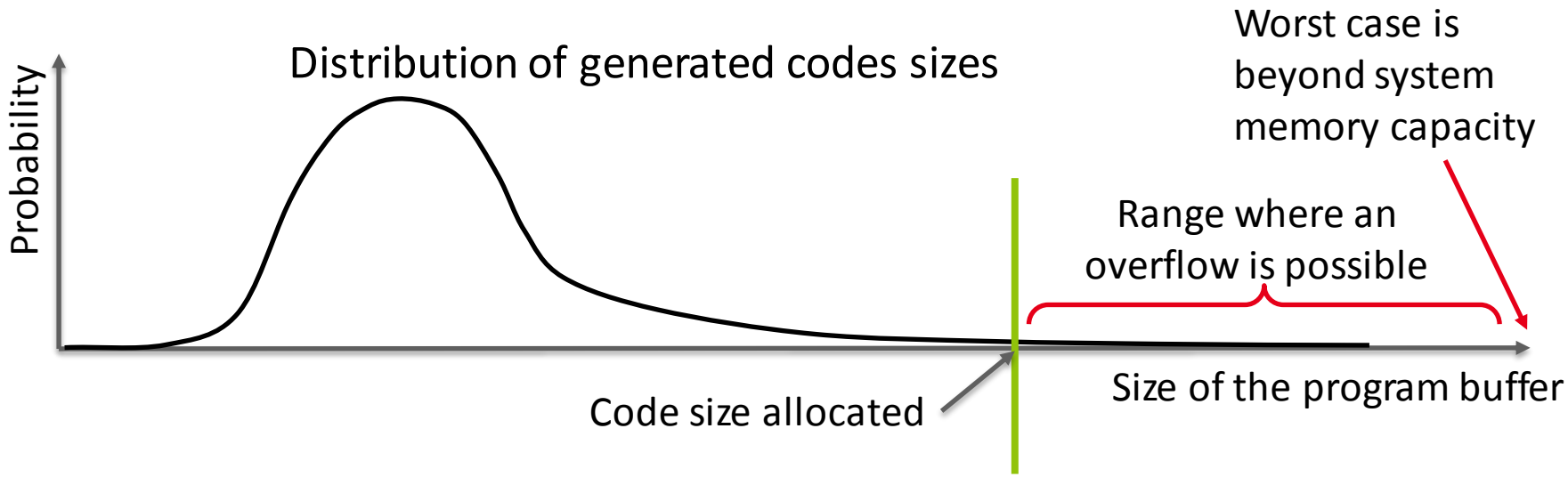
Polymorphic version, with COGITO:



MEMORY ALLOCATION OF CODE BUFFERS

Challenges

- No Operating System, no dynamic memory allocation (malloc), no MPU
- Generated code has a variable size
- Largest possible code size does not fit in system memory



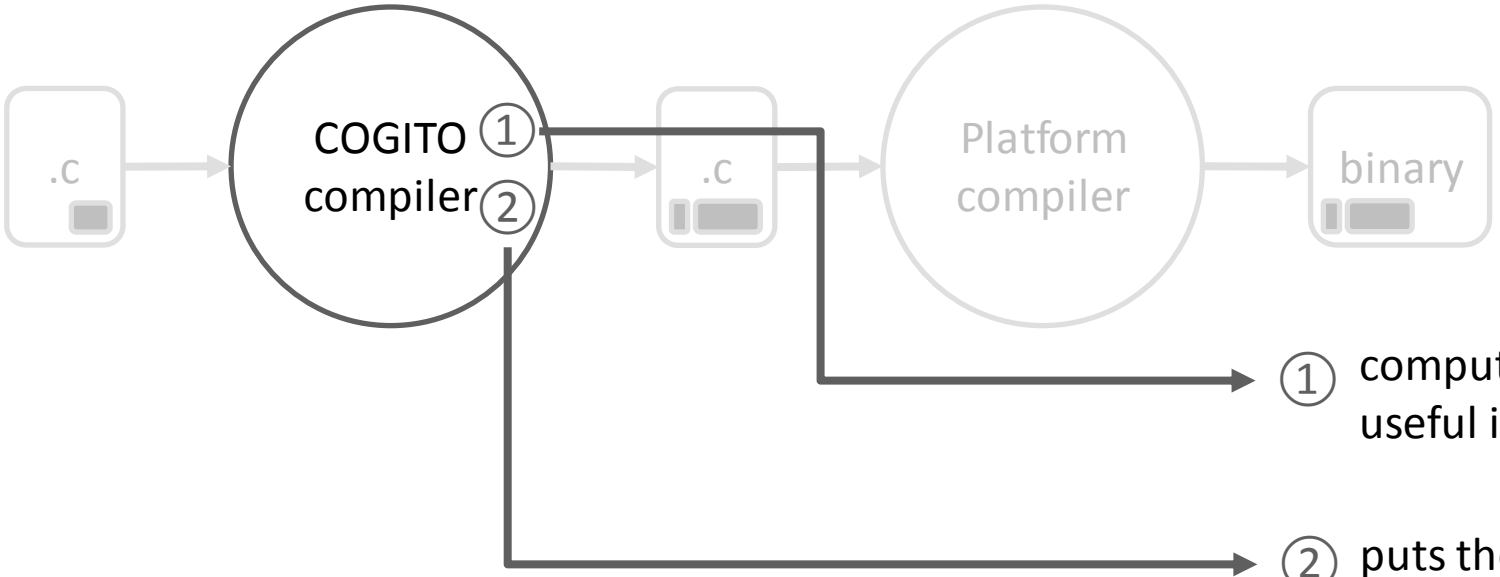
Idea: compute a realistic code size suitable for (1-p) code generations.

- Threshold p: probability of memory overflow
- $p = 10^{-6}$ by default (configurable)
- Computation of the code size done automatically by the compiler

For a 100 instructions code (low config.), allocated size is **5x** smaller than worst case!

PREVENTION OF CODE BUFFER OVERFLOWS

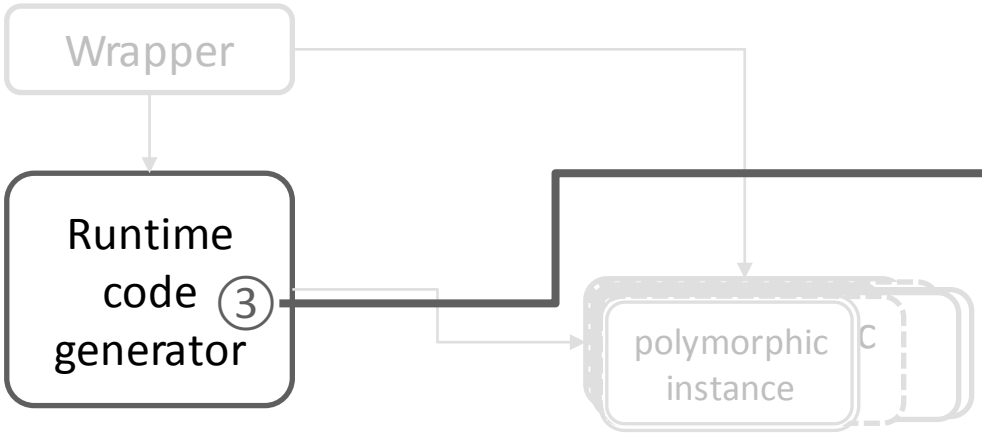
STATICALLY



1 computes the size of useful instructions

2 puts the information directly in runtime code generator's code

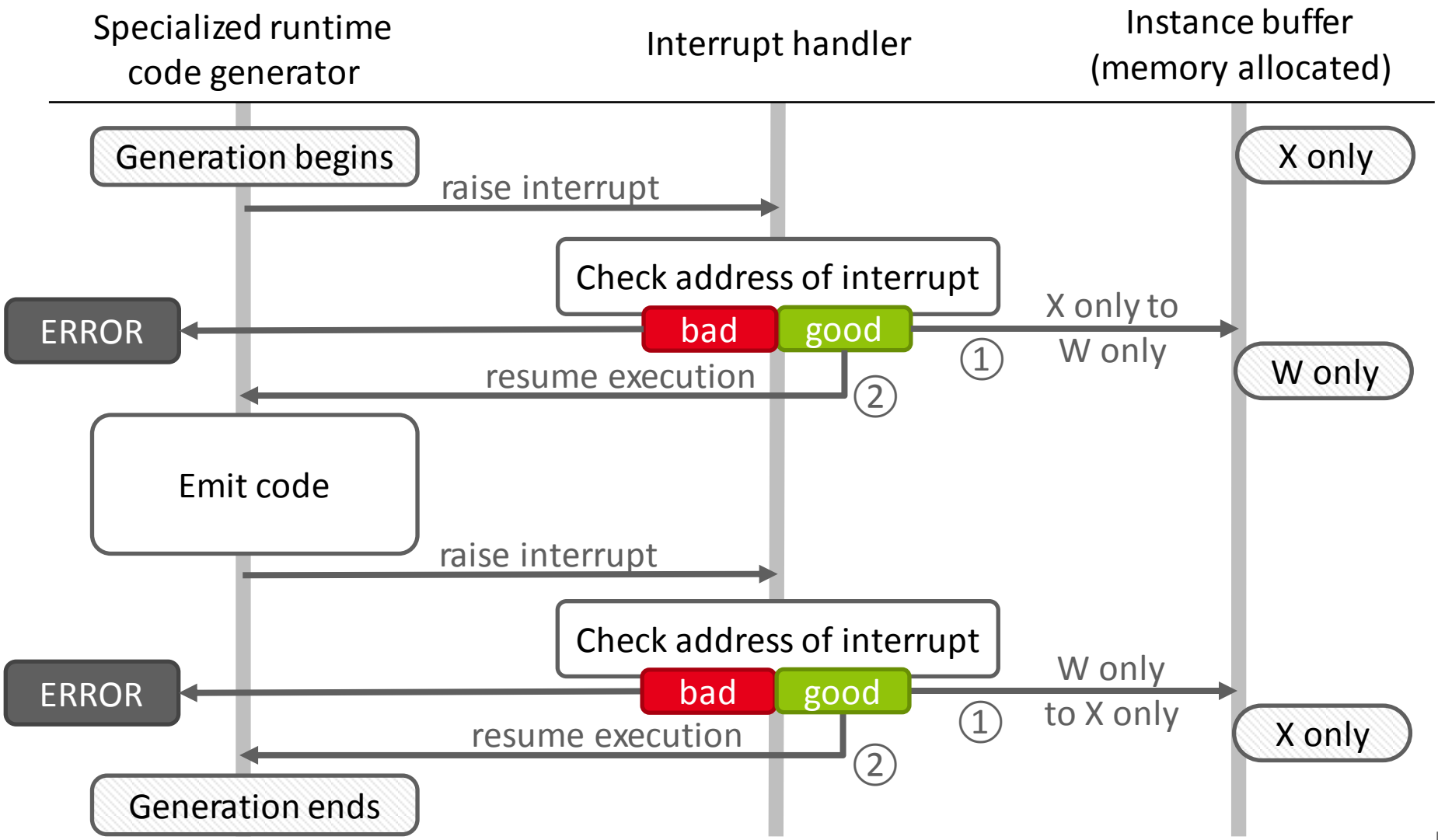
RUNTIME



3 always keep space for useful instructions (limit polymorphism if necessary)

MANAGEMENT OF MEMORY PERMISSIONS

Objective: Guarantee W ⊕ X and that only the generator can write into the buffer

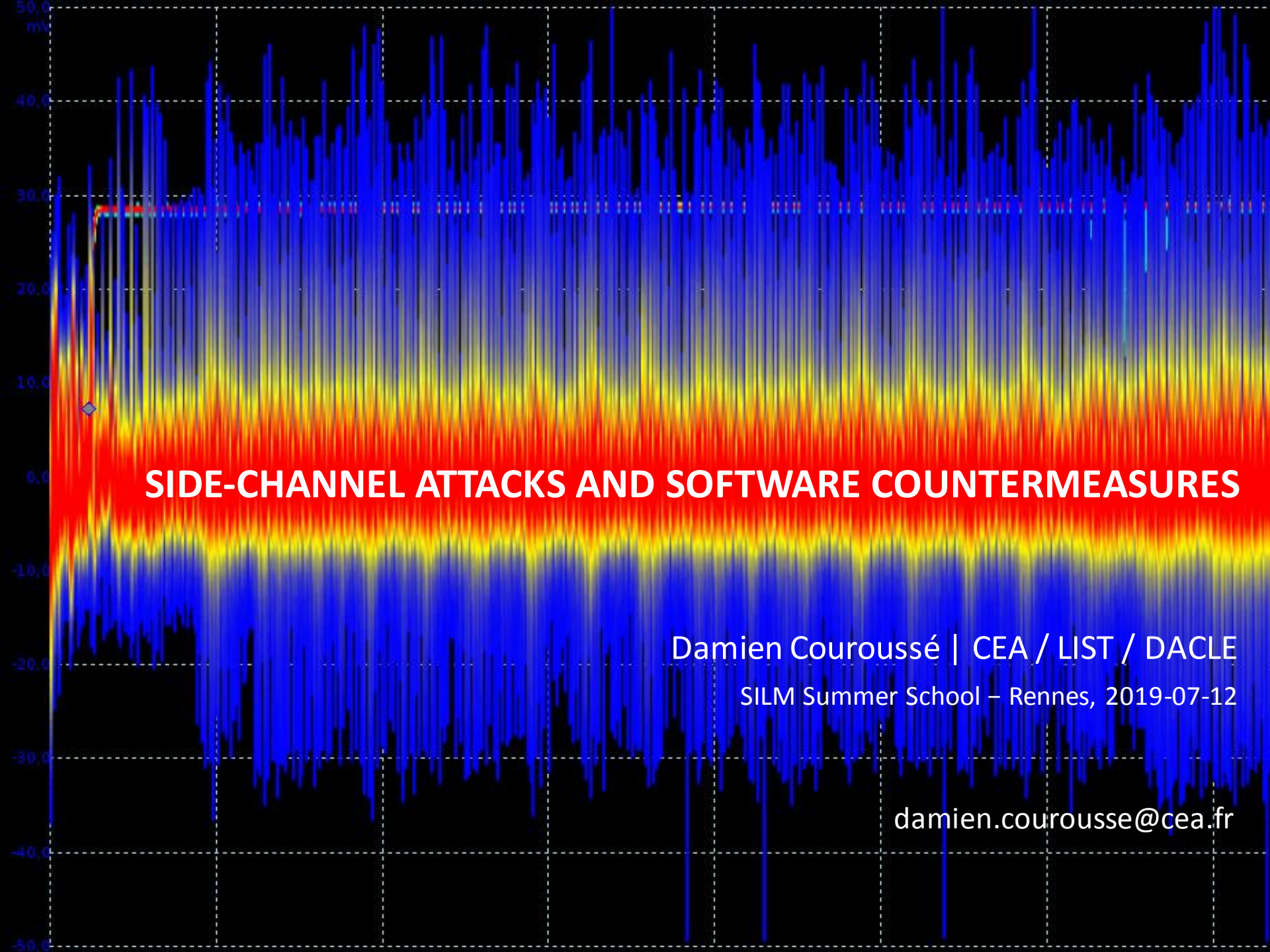


CONCLUSION

- **Leverage the compiler to implement counter-measures**
  - Automation, flexibility, configurability
- **Leverage compiler analysis and compiler optimisations to improve the effectiveness of counter-measures**
- **Verification tools are needed *after* the compilation of countermeasures**

Ongoing directions

- **Hardware security with software-only counter-measures is ~~impossible~~ challenging**
  - Challenge your threat model
  - HW/SW co-design of countermeasures



SIDE-CHANNEL ATTACKS AND SOFTWARE COUNTERMEASURES

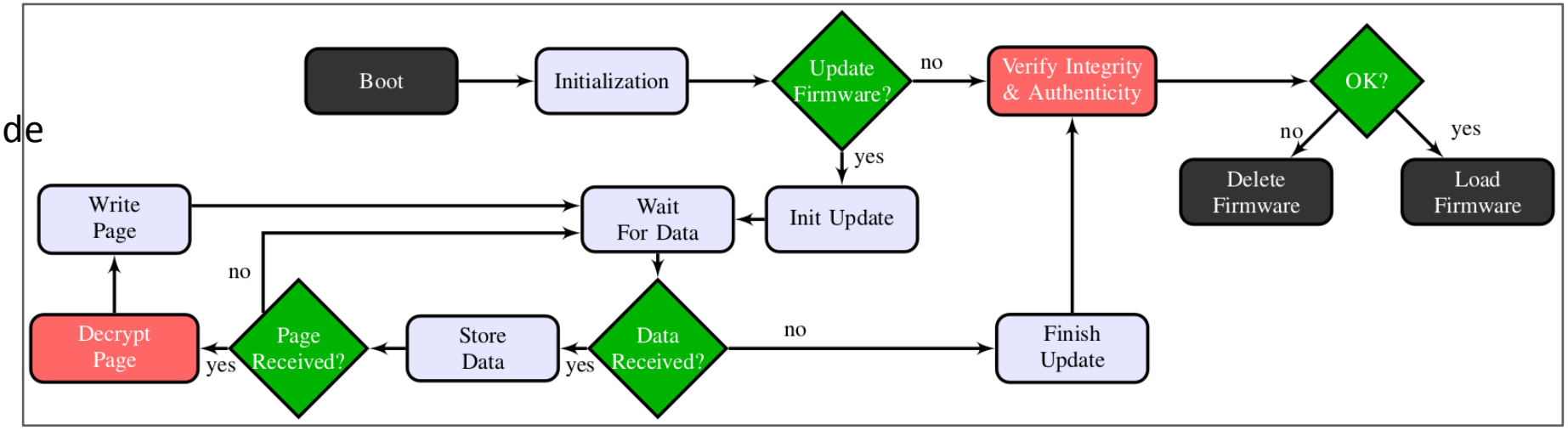
Damien Couroussé | CEA / LIST / DACLE

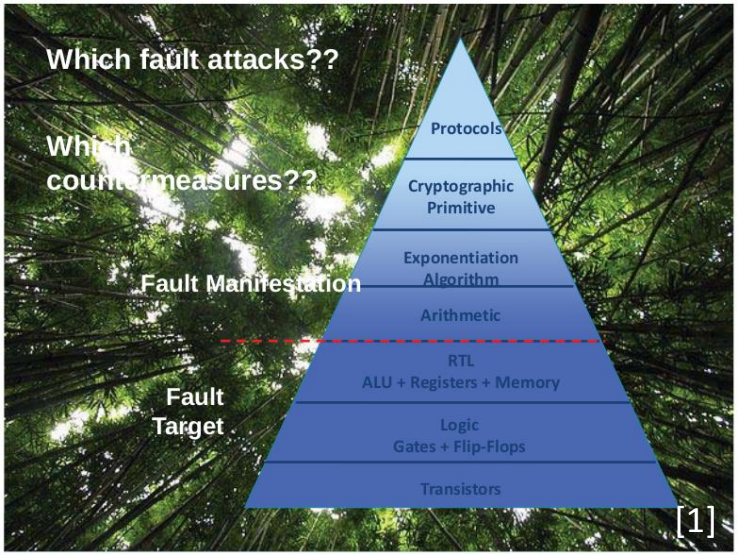
SILM Summer School – Rennes, 2019-07-12

damien.courousse@cea.fr

FAULT INJECTION ATTACKS: APPLICATION TO BFUS

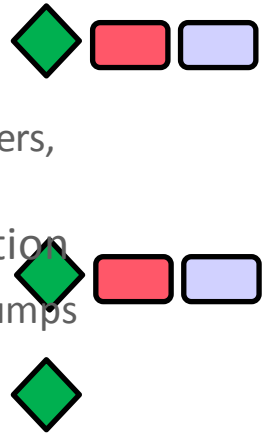
slide





Fault models, at the Instruction Set Architecture (ISA) level:

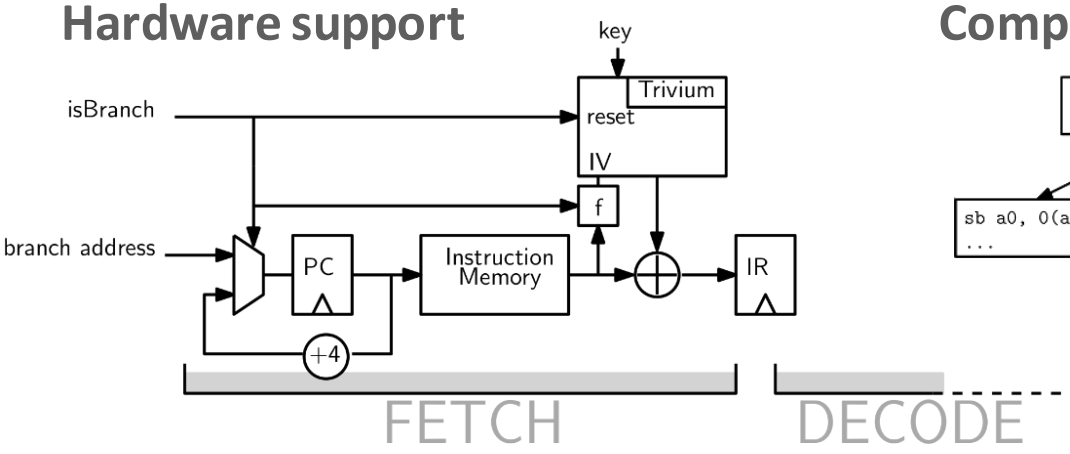
1. Data alteration, down to the bit level.
   - ROM / RAM, processor registers
   - Bit flip, bit stuck-at
   - Typically: modification of loop counters, crypto data, opcode corruption.
2. Instruction skip, instruction modification
   - Typically: NOP execution, arbitrary jumps
3. Modification of the control flow, e.g., test inversion



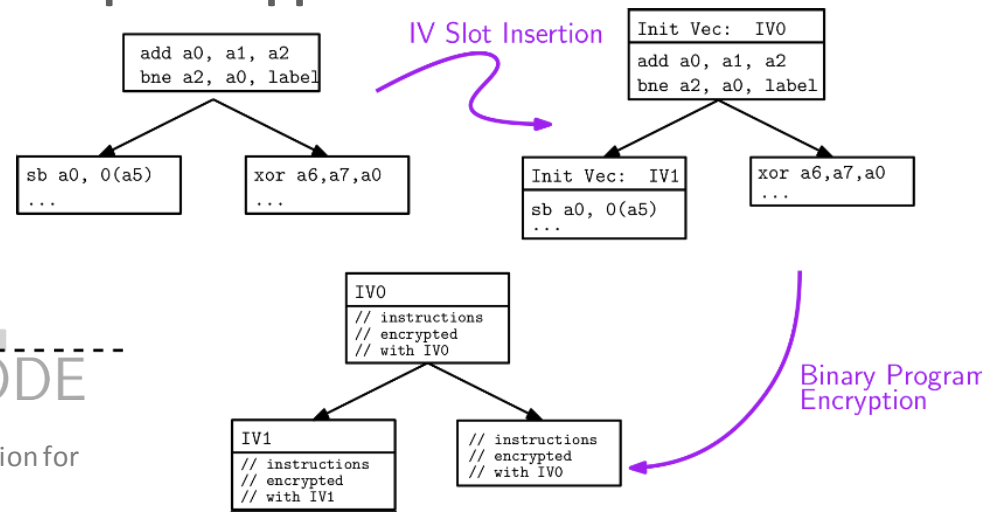
[1] I. Polian, M. Joye, I. Verbauwhede, M. Witteman, and J. Heyszl, 'Controlled fault injection: wishful thinking, thoughtful engineering or just luck ?', FDTC, 2017.

PROGRAM ENCRYPTION

Hardware support



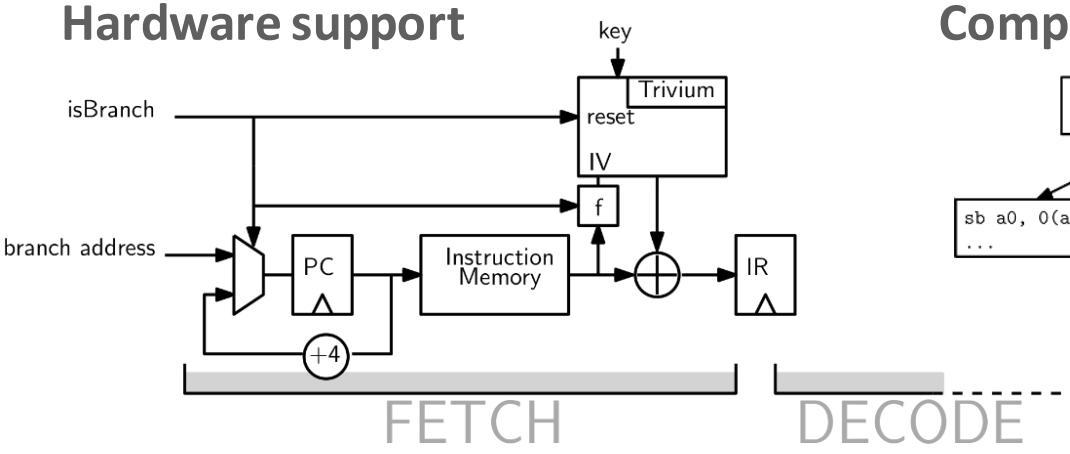
Compiler support



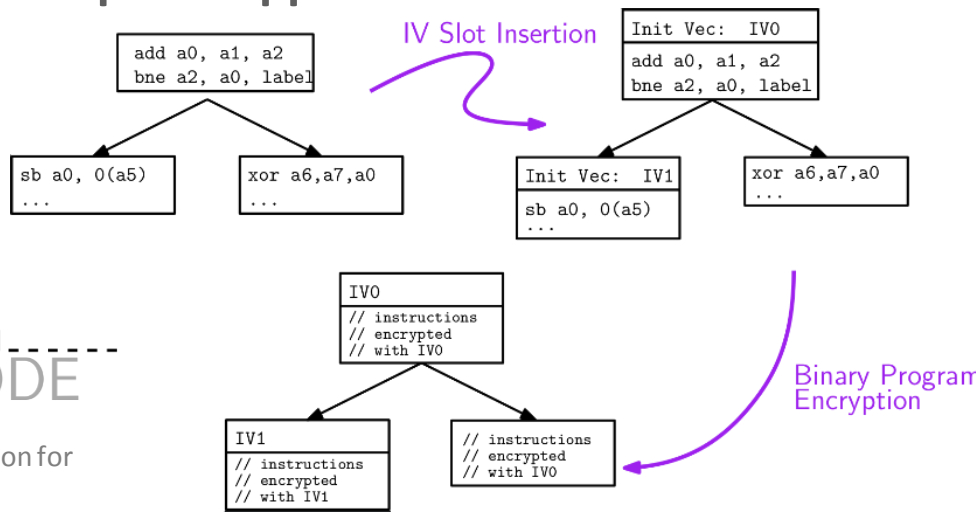
Hiscock, T., Savry, O. and Goubin, L. (2017) 'Lightweight Software Encryption for Embedded Processors' Euromicro DSD

POLYMORPHIC EXECUTION OF ENCRYPTED PROGRAMS

Hardware support



Compiler support

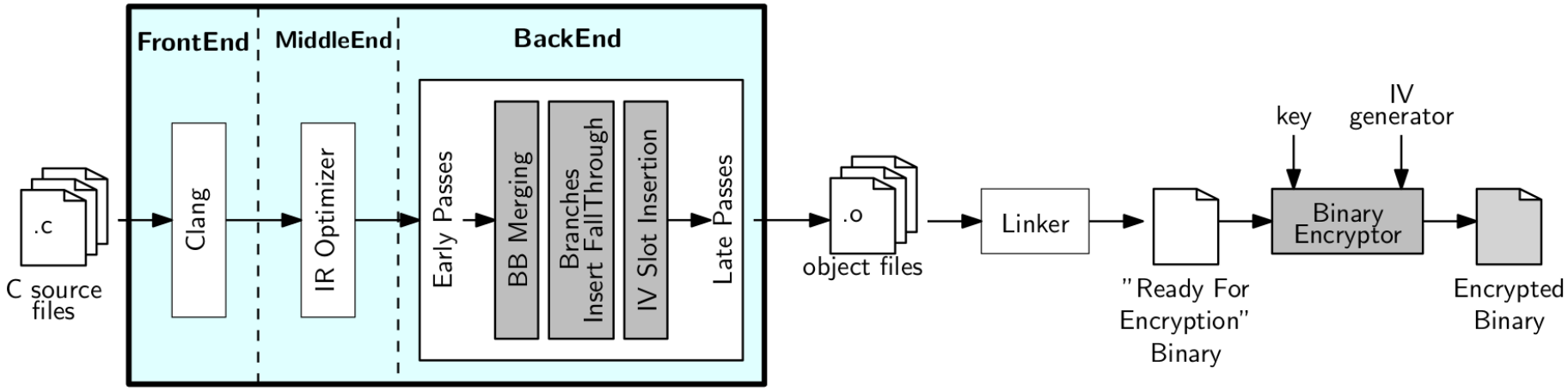


Hiscock, T., Savry, O. and Goubin, L. (2017) 'Lightweight Software Encryption for Embedded Processors' Euromicro DSD

	Side-channel attacks	Static reverse-engineering	Dynamic reverse (SCARE)
Encrypted program	✗	✓	✗
Code polymorphism	✓	✗	(✓)
Encryption + polymorphism	✓	✓	✓

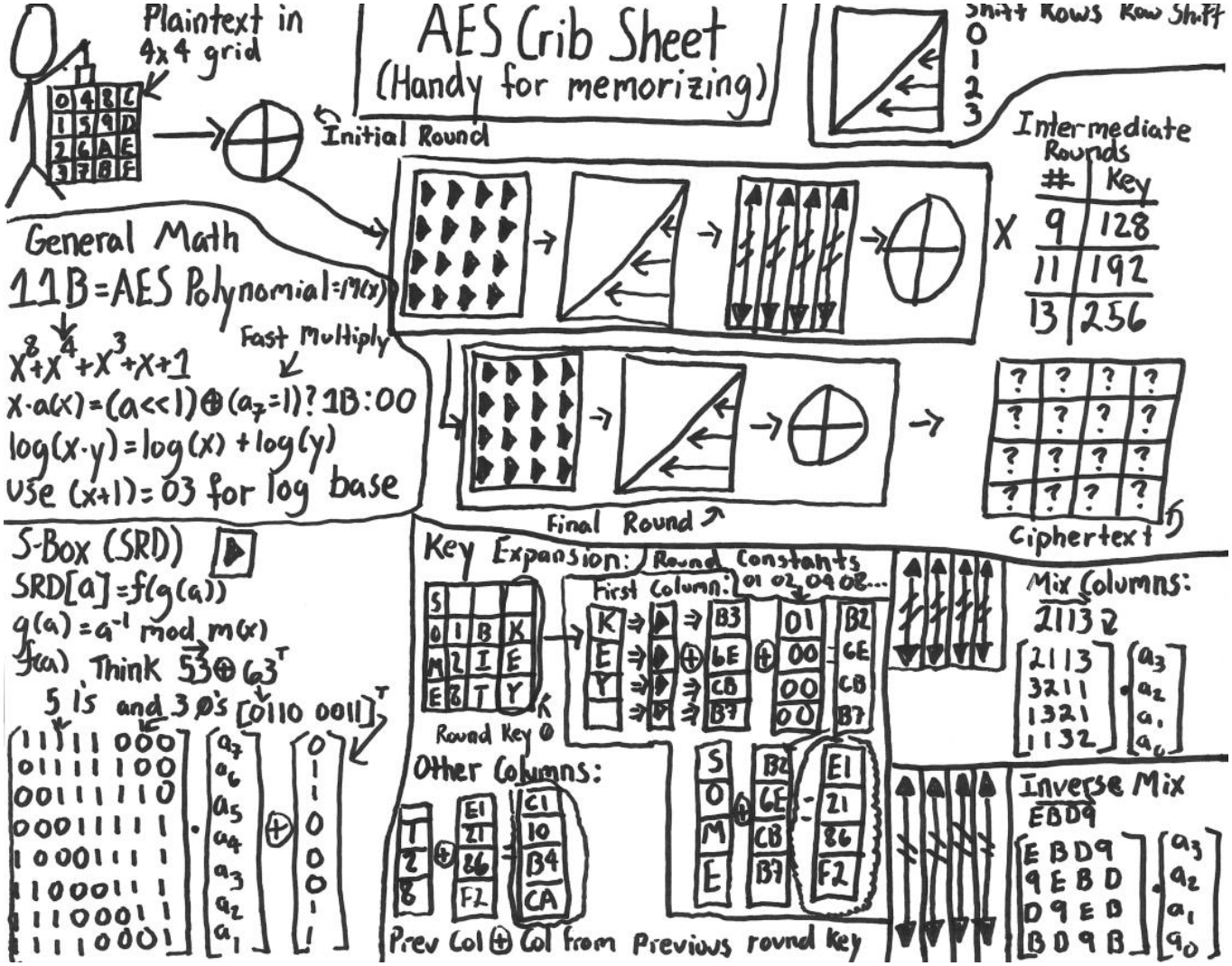
POLYMORPHIC EXECUTION OF ENCRYPTED PROGRAMS

Toolchain support

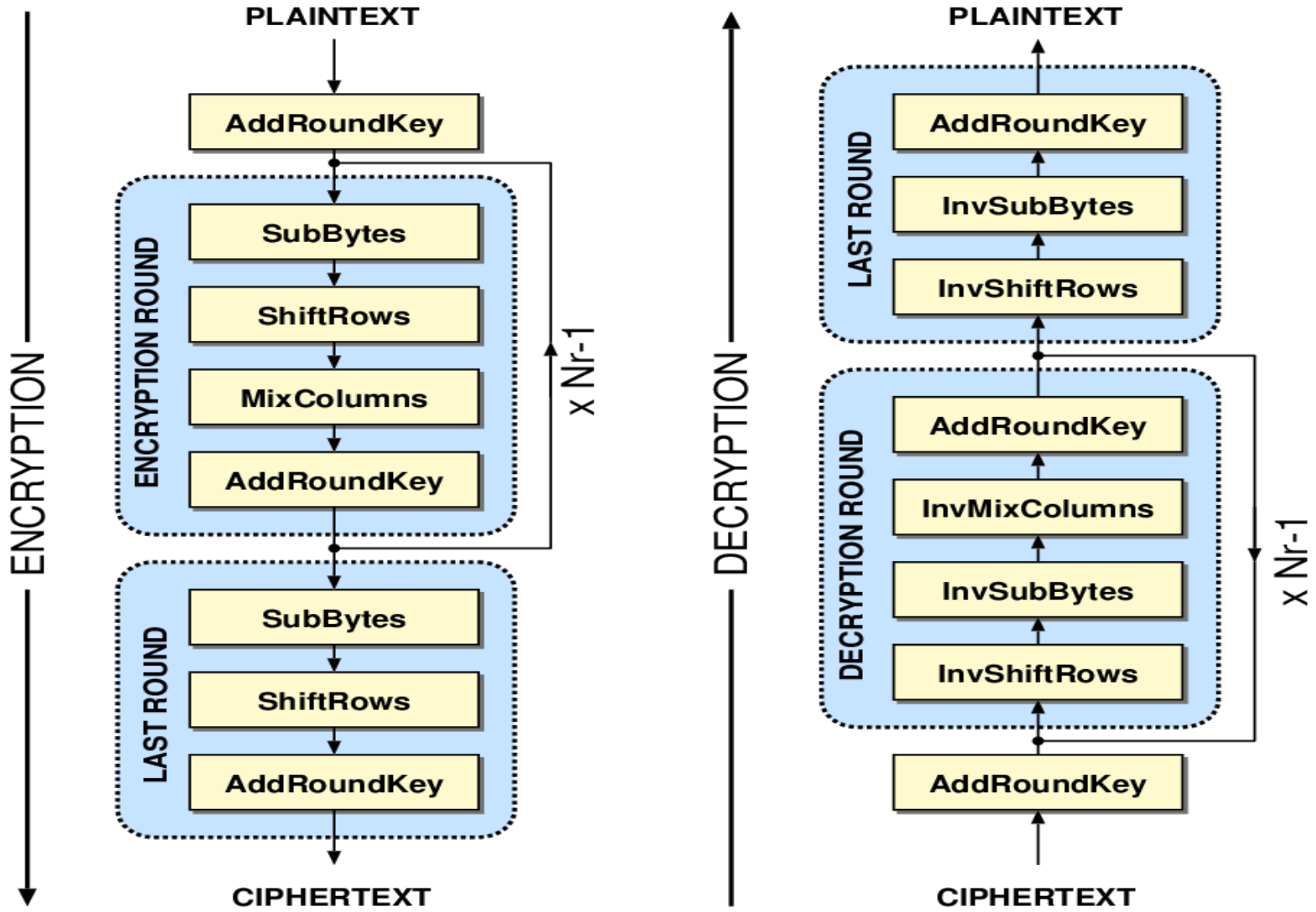


	Side-channel attacks	Static reverse-engineering	Dynamic reverse (SCARE)
Encrypted program	✗	✓	✗
Code polymorphism	✓	✗	(✓)
Encrypted code polymorphism	✓	✓	✓

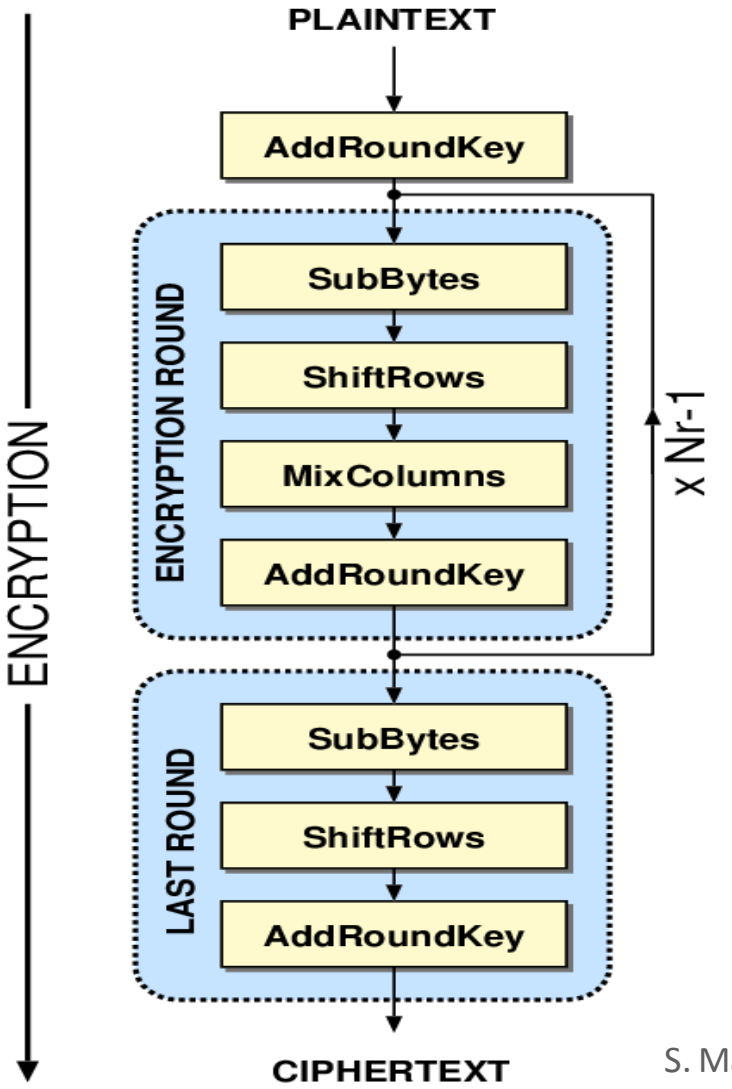
- Our toolchain now targets **RISCV ISA** (+ARMv7)
- **llvm-RISCV** back-end generating **encryption-ready** binaries
- Standalone **binary encryptor**
- **HW decryption** added to the **Spike** Instruction Set Simulator
- Currently working on the encryption of polymorphic instances

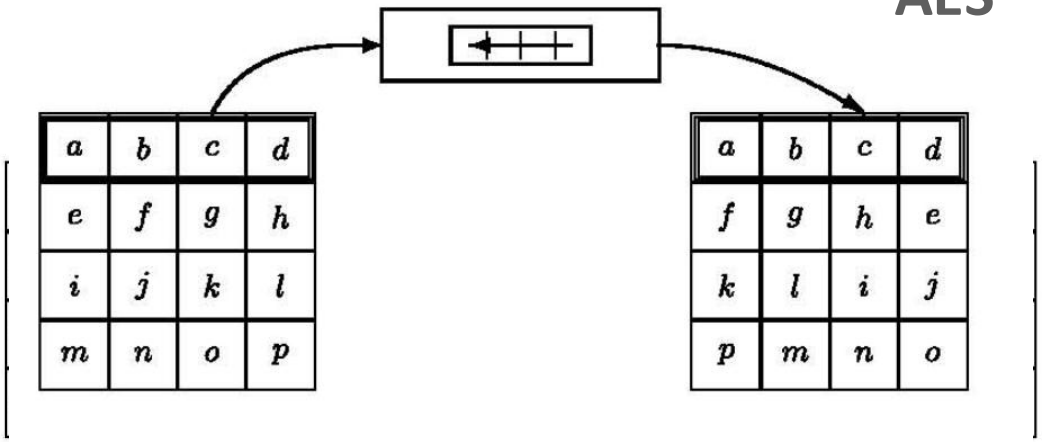

AES Crib Sheet
(Handy for memorizing)
Plaintext in 4x4 grid
Initial Round
Shift Rows Row Shift
0
1
2
3
Intermediate Rounds
Key
9 128
11 192
13 256
X
General Math
11B = AES Polynomial = m(x)
$X^8+X^4+X^3+X+1$
Fast Multiply
$x \cdot a(x) = (a<<1) \oplus (a_7==1)\ ?\ 1B : 00$
$\log(x \cdot y) = \log(x) + \log(y)$
use (x+1) = 03 for log base
Final Round
Ciphertext
S-Box (SRD)
SRD[a] = f(g(a))
$g(a) = a^{-1} \bmod m(x)$
f(a) Think $53 \oplus 63^T$
5 1's and 3 0's [0110 0011]T
Key Expansion:
Round Constants
01 02 04 08...
First Column:
Round Key 0
Other Columns:
Prev Col ⊕ Col from Previous round key
Mix Columns:
2113 2
Inverse Mix
EBD9

AES, TIME AFTER TIME (BUT SO USEFUL…)



AES



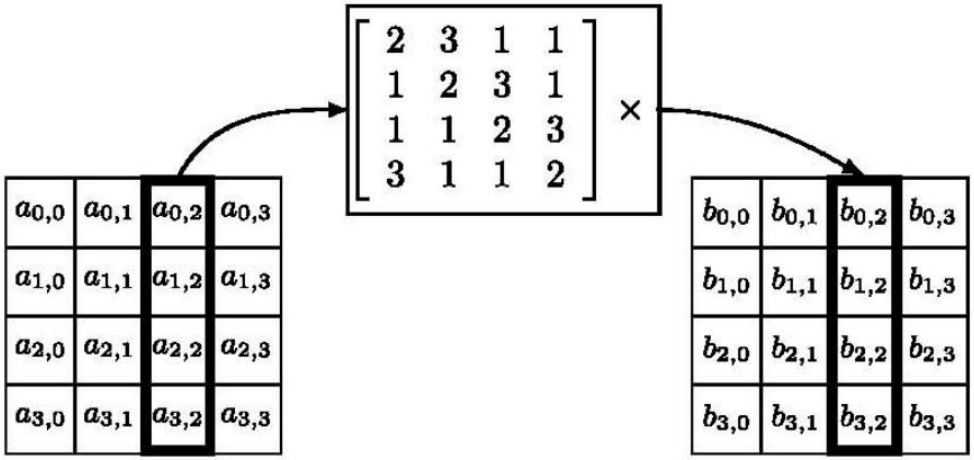


a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

a	b	c	d
f	g	h	e
k	l	i	j
p	m	n	o

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

Fig ...on.



$$\begin{bmatrix} 2 & 3 & 1 & 1 \\ 1 & 2 & 3 & 1 \\ 1 & 1 & 2 & 3 \\ 3 & 1 & 1 & 2 \end{bmatrix} \times$$

$a_{0,0}$	$a_{0,1}$	$a_{0,2}$	$a_{0,3}$
$a_{1,0}$	$a_{1,1}$	$a_{1,2}$	$a_{1,3}$
$a_{2,0}$	$a_{2,1}$	$a_{2,2}$	$a_{2,3}$
$a_{3,0}$	$a_{3,1}$	$a_{3,2}$	$a_{3,3}$

$b_{0,0}$	$b_{0,1}$	$b_{0,2}$	$b_{0,3}$
$b_{1,0}$	$b_{1,1}$	$b_{1,2}$	$b_{1,3}$
$b_{2,0}$	$b_{2,1}$	$b_{2,2}$	$b_{2,3}$
$b_{3,0}$	$b_{3,1}$	$b_{3,2}$	$b_{3,3}$

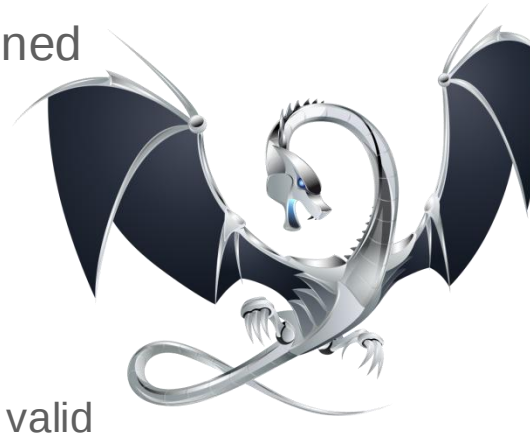
Figure B.7. **MixColumns operates on the columns of the state.**

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

STANDARD COMPILERS AND SECURITY

COMPILER DUTIES & OBJECTIVES

- **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)
    - Undefined behaviours?
  - No formal assurance
    - Except few works, such as CompCert
  - Correctness by construction?
    - The source code written by the developper 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…

COMPILER RIGHTS

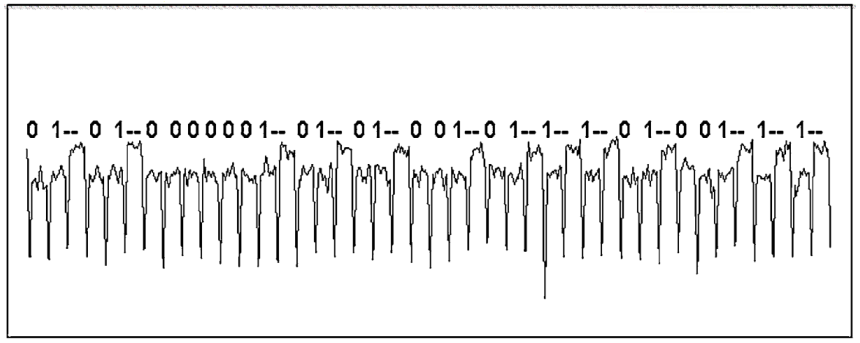
- **Rights**
  - Reorganise contents of the target program, as long as program semantics is preserved
    - Machine instructions, basic blocs
  - Select the best translation for a source code operation / instruction
  - Remove parts of the program, as long as the program functionality is considered to 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 utilitary passes
  - … backends, etc.

Will break your security mechanisms

EXPLOITATION OF SIDE-CHANNEL INFORMATION LEAKAGE

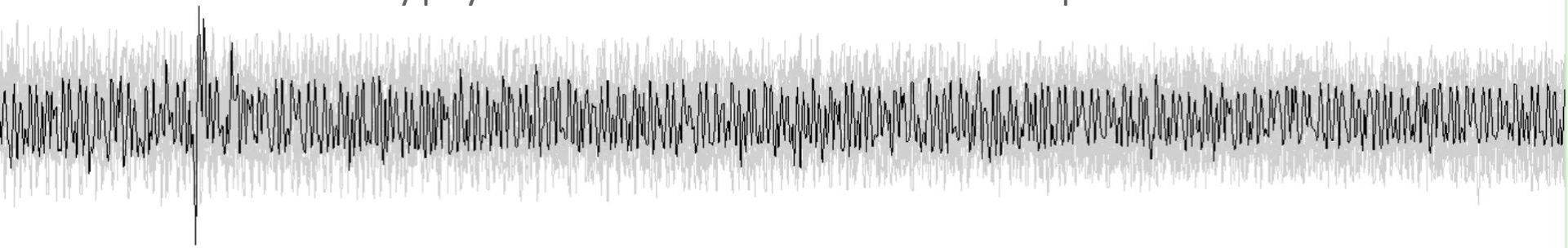


Simple power analysis (SPA)

SPA leaks from an RSA implementation

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.

Correlation Power/EM Analysis (CPA/CEMA) – Can be generalised to any physical observation of the secured computation

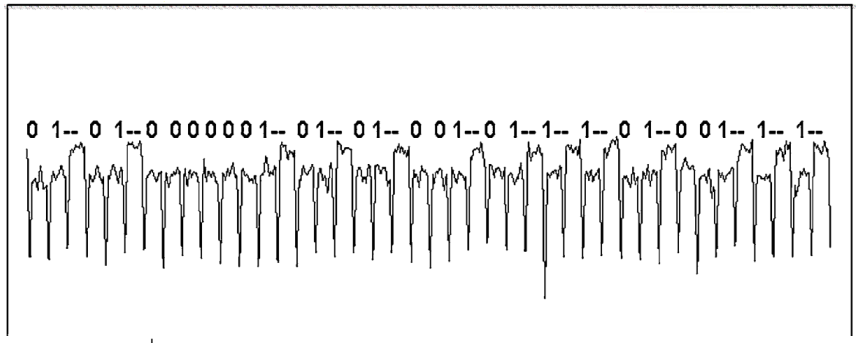


Key found!

- AES, unprotected implementation
- EM traces
- Attack on the output of the 1st SBOX

After the encryption of 4240 Bytes of data!

EXPLOITATION OF SIDE-CHANNEL INFORMATION LEAKAGE

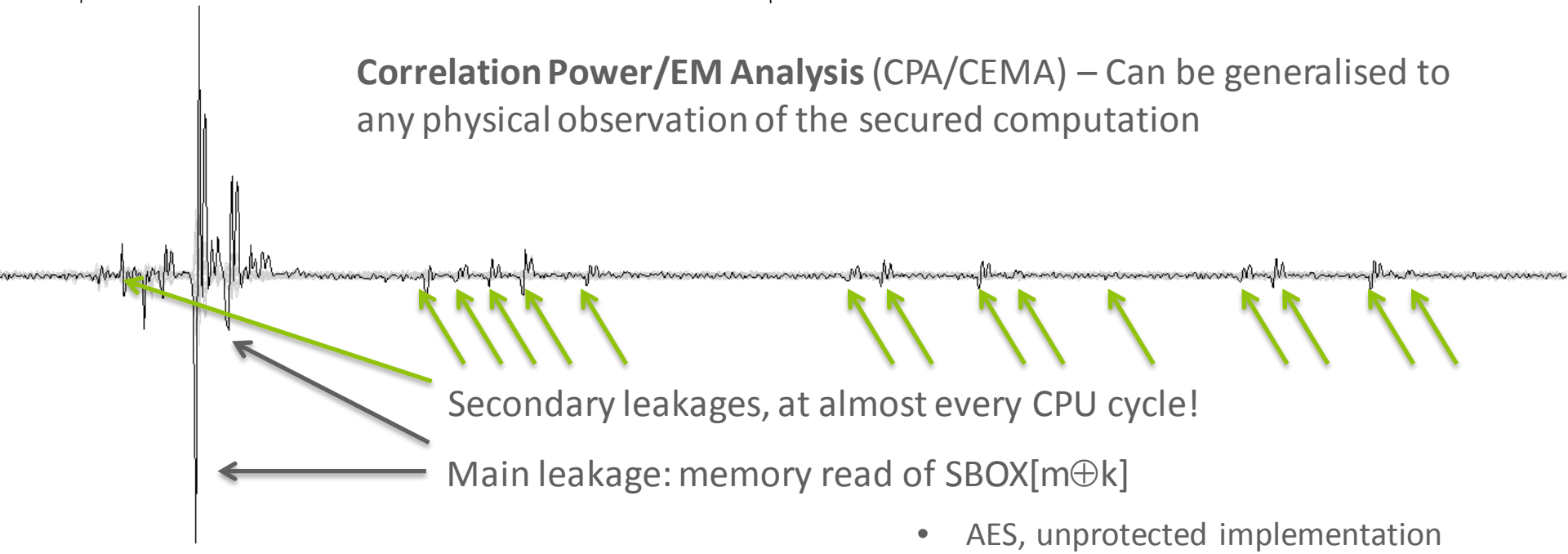


Simple power analysis (SPA)

SPA leaks from an RSA implementation

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.

Correlation Power/EM Analysis (CPA/CEMA) – Can be generalised to any physical observation of the secured computation



Secondary leakages, at almost every CPU cycle!

Main leakage: memory read of SBOX[m⊕k]

- AES, unprotected implementation
- EM traces
- Attack on the output of the 1st SBOX