Nicolas Belleville
Damien Couroussé
Karine Heydemann
Henri-Pierre Charles

AUTOMATED SOFTWARE PROTECTION FOR THE MASSES AGAINST SIDE-CHANNEL ATTACKS

INTRODUCTION

- **Focus on power/EM based side channel attacks**

- **Objective: solution usable by anybody (not only security experts) on any code (not only block ciphers)**

- **Software countermeasures**
  - Masking
    - Automated masking for ANY code is hard
    - Masking scheme depends on underlying code
    - Hard to be efficient in terms of execution time for any code
  - **Hiding**
    - Generic principle
    - Do not remove leakage, but make it harder to exploit

INTRODUCTION

- **Focus on power/EM based side channel attacks**
- **Objective: solution usable by anybody (not only security experts) on any code (not only block ciphers)**
- **Software countermeasures**
  - Masking
    - Automated masking for ANY code is hard
    - Masking scheme depends on underlying code
    - Hard to be efficient in terms of execution time for any code
  - **Hiding**
    - Generic principle
    - Do not remove leakage, but make it harder to exploit

> Code polymorphism:
> ability to **change the observable behaviour** of a software component **without changing** its **functional properties**

CODE POLYMORPHISM USING SPECIALISED DYNAMIC CODE GENERATION

- **Objective: making the executed code vary**
  → use a generator to regenerate the code regularly
  - Performs **code transformations** guided by **randomness**
  - Produces a different code at every generation

- **Generators are specialized**
  - Each function to be secured has its **own generator**

- **A generator works on an assembly-level representation of the function**
  - Code transformations related to this representation:
  - Instructions shuffling
  - Register shuffling
  - Semantic equivalent
  - Insertion of noise instructions

PROBLEMS TO ANSWER

- **How to write a generator?**
- **Runtime code generation is usually expensive**
  - Is specialization capable of lowering the cost?
- **Runtime code generation needs W and X permissions**
- **Code size varies from one generation to another**
  - Semantic equivalent
  - Insertion of noise instructions

AUTOMATIC APPLICATION OF CODE POLYMORPHISM

- **Objective:**
  - Start from a C file
  - Produce a new C file with polymorphism countermeasure applied to target functions

Main idea:
For each targetted function:
- get a sequence of instructions
- construct a generator from that
- modify the sequence of instructions dynamically

AUTOMATIC APPLICATION OF CODE POLYMORPHISM

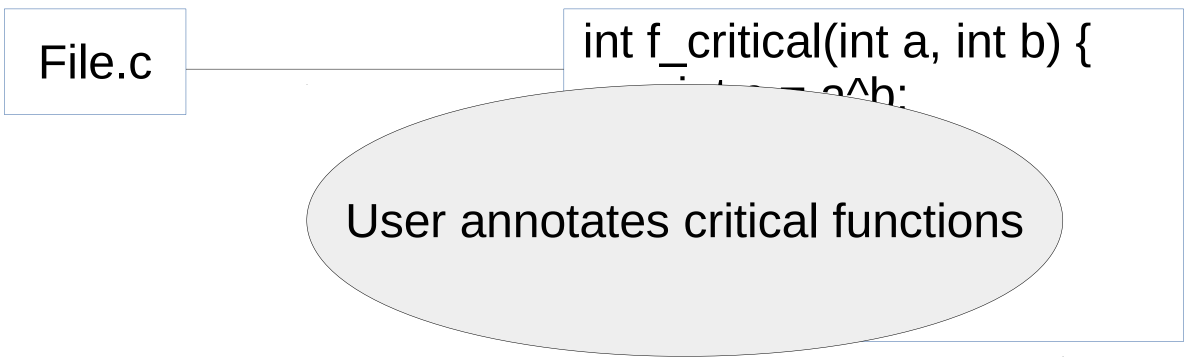
- **Objective:**
  - Start from a C file
  - Produce a new C file with polymorphism countermeasure applied to target functions

File.c

```
int f_critical(int a, int b) {
	int c = a^b;
	a = a+b;
	a = a % c;
	return a;
}
```

AUTOMATIC APPLICATION OF CODE POLYMORPHISM

- **Objective:**
  - Start from a C file
  - Produce a new C file with polymorphism countermeasure applied to target functions



AUTOMATIC APPLICATION OF CODE POLYMORPHISM

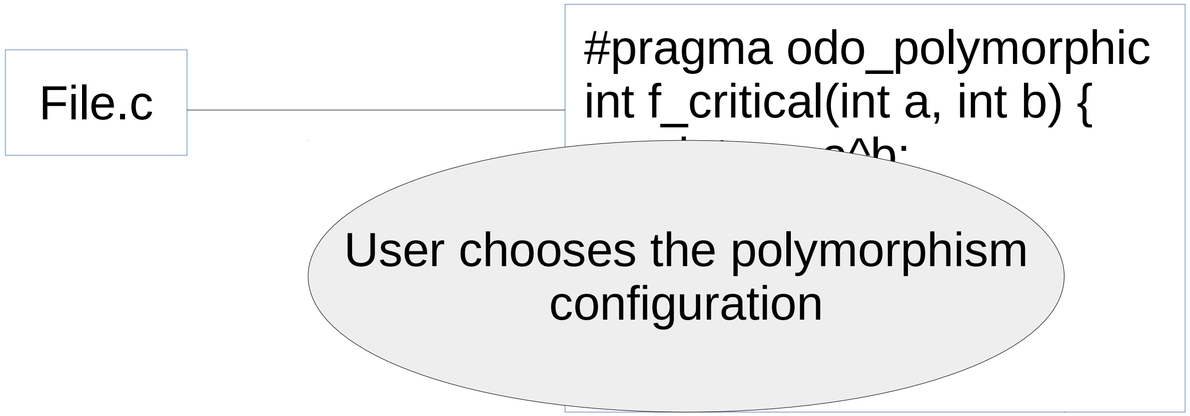
- **Objective:**
  - Start from a C file
  - Produce a new C file with polymorphism countermeasure applied to target functions

File.c

```
#pragma odo_polymorphic
int f_critical(int a, int b) {
	int c = a^b;
	a = a+b;
	a = a % c;
	return a;
}
```

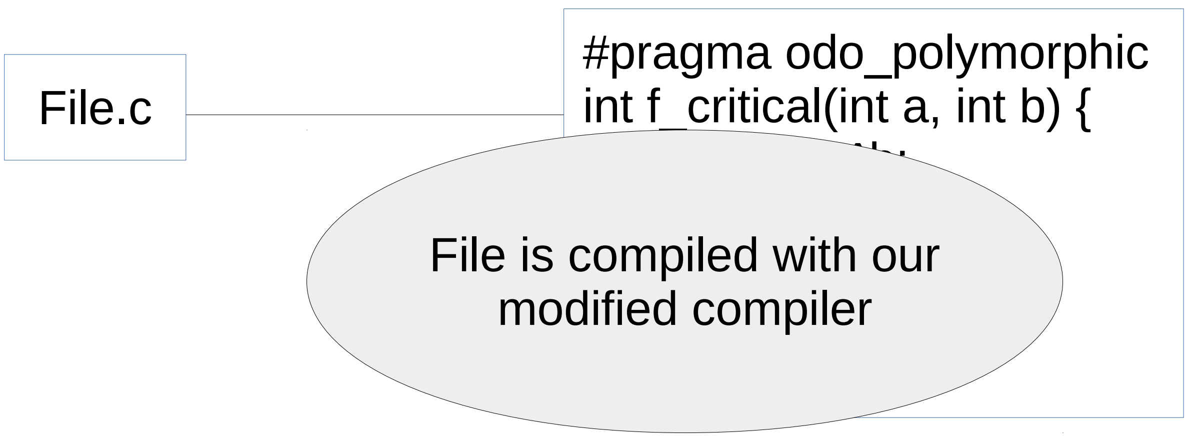
AUTOMATIC APPLICATION OF CODE POLYMORPHISM

- **Objective:**
  - Start from a C file
  - Produce a new C file with polymorphism countermeasure applied to target functions



AUTOMATIC APPLICATION OF CODE POLYMORPHISM

- **Objective:**
  - Start from a C file
  - Produce a new C file with polymorphism countermeasure applied to target functions



AUTOMATIC APPLICATION OF CODE POLYMORPHISM

- **Objective:**
  - Start from a C file
  - Produce a new C file wi… …rget functions

File.c

```
code code_f[CODE_SIZE];
void SGPC_f_critical() {
	raise_interrupt_rm_X_add_W(code_f);
	reg_t r[] = {0,1,2,3,4,5,6,...,12,13,14,15};
	push_T2_callee_saved_registers();
	eor_T2(r[4], r[1], r[0]);
	add_T2(r[0], r[1], r[0]);
	sdiv_T2(r[1], r[0], r[4]);
	mls_T2(r[0], r[1], r[4], r[0]);
	pop_T2_callee_saved_registers();
	raise_interrupt_rm_W_add_X(code_f);
}
int f_critical(int a, int b) {
	if (SHOULD_BE_REGENERATED())
		SGPC_f_critical();
	return code_f(a, b);
}
```

AUTOMATIC APPLICATION OF CODE POLYMORPHISM

- **Objective:**
  - Start from a C file
  - Produce a new C file wi ... rget functions

File.c

```
code code_f[CODE_SIZE];
void SGPC_f_critical() {
	raise_interrupt_rm_X_add_W(code_f);
	reg_t r[] = {0,1,2,3,4,5,6,...,12,13,14,15};
	push_T2_callee_saved_registers();
	eor_T2(r[4], r[1], r[0]);
	add_T2(r[0], r[1], r[0]);
	sdiv_T2(r[1], r[0], r[4]);
	mls_T2(r[0], r[1], r[4], r[0]);
	pop_T2_callee_saved_registers();
	raise_interrupt_rm_W_add_X(code_f);
}
int f_critical(int a, int b) {
	if (SHOULD_BE_REGENERATED())
		SGPC_f_critical();
	return code_f(a, b);
}
```

AUTOMATIC APPLICATION OF CODE POLYMORPHISM

- **Objective:**
  - Start from a C file
  - Produce a new C file wi rget functions

File.c

```
code code_f[CODE_SIZE];
void SGPC_f_critical() {
	raise_interrupt_rm_X_add_W(code_f);
	reg_t r[] = {0,1,2,3,4,5,6,...,12,13,14,15};
	push_T2_callee_saved_registers();
	eor_T2(r[4], r[1], r[0]);
	add_T2(r[0], r[1], r[0]);
	sdiv_T2(r[1], r[0], r[4]);
	mls_T2(r[0], r[1], r[4], r[0]);
	pop_T2_callee_saved_registers();
	raise_interrupt_rm_W_add_X(code_f);
}
int f_critical(int a, int b) {
	if (SHOULD_BE_REGENERATED())
		SGPC_f_critical();
	return code_f(a, b);
}
```

AUTOMATIC APPLICATION OF CODE POLYMORPHISM

- **Objective:**
  - Start from a C file
  - Produce a new C file wi… …rget functions

File.c

```
code code_f[CODE_SIZE];
void SGPC_f_critical() {
	raise_interrupt_rm_X_add_W(code_f);
	reg_t r[] = {0,1,2,3,4,5,6,...,12,13,14,15};
	push_T2_callee_saved_registers();
	eor_T2(r[4], r[1], r[0]);
	add_T2(r[0], r[1], r[0]);
	sdiv_T2(r[1], r[0], r[4]);
	mls_T2(r[0], r[1], r[4], r[0]);
	pop_T2_callee_saved_registers();
	raise_interrupt_rm_W_add_X(code_f);
}
int f_critical(int a, int b) {
	if (SHOULD_BE_REGENERATED())
		SGPC_f_critical();
	return code_f(a, b);
}
```

AUTOMATIC APPLICATION OF CODE POLYMORPHISM

- **Objective:**
  - Start from a C file
  - Produce a new C file wi… …rget functions

File.c

```
code code_f[CODE_SIZE];
void SGPC_f_critical() {
	raise_interrupt_rm_X_add_W(code_f);
	reg_t r[] = {0,1,2,3,4,5,6,...,12,13,14,15};
	push_T2_callee_saved_registers();
	eor_T2(r[4], r[1], r[0]);
	add_T2(r[0], r[1], r[0]);
	sdiv_T2(r[1], r[0], r[4]);
	mls_T2(r[0], r[1], r[4], r[0]);
	pop_T2_callee_saved_registers();
	raise_interrupt_rm_W_add_X(code_f);
}
int f_critical(int a, int b) {
	if (SHOULD_BE_REGENERATED())
		SGPC_f_critical();
	return code_f(a, b);
}
```

CODE TRANSFORMATIONS USED AT RUNTIME

- **Register shuffling**
  - Permutation among all equivalent registers

- **Instruction shuffling**
  - Shuffling of independent instructions (use/def register analysis)

- **Use of semantic equivalent**
  - Random choice between sequences of instructions equivalent to the original instructions
  - Semantic equivalents available for a limited number of instructions
  - Ex: a xor b <=> (a xor r) xor (b xor r)

- **Insertion of noise instructions**
  - Useless instructions among frequently used ones (xor, sub, load, add)
  - A probability model determines the number of noise instructions to be inserted (possibly 0)
  - One insertion in between each pair of original instructions

REMAINING PROBLEMS

- **Memory write and execute permissions**
  - Code generation → both write and execute permissions on a memory segment → could be exploited to mount an attack
- **Code size varies**
  - Allocated memory should be large enough
  - But not too large!

MANAGEMENT OF MEMORY PERMISSIONS

- **W and X permissions required for dynamic code generation**
- **Use the specialisation of generator to change permissions**
- **For each secured function, only one generator allowed to write in allocated buffer**
- **Interrupt raised to change memory permissions between W only and X only**
  - When generation begins: X only to W only
  - When generation ends: W only to X only
  - Interrupt handler knows which generator is associated with which memory zone

STATIC ALLOCATION OF A REALISTIC SIZE

- **How to determine a realistic size for allocation?**
  - Worst case is terrible and never happens in programs long enough

    → need for a better metric
  - Worst case used for semantic equivalents only:
    - Size of longest semantic equivalent
    - e.g. if (a xor b) can be replaced by (a xor r) xor (b xor r), we reserve space for 3 xor instructions
  - For insertion of noise instruction:
    - MSD (Maximum Standard Deviation == mean+standard deviation) is used
    - Better than mean: mean would only work for infinitely long programs
    - Better than worst case

STATIC ALLOCATION OF A REALISTIC SIZE

- **How to determine a realistic size for allocation?**
  - Worst case is terrible and never happens in programs long enough

    → need for a better metric
  - Worst case used for semantic equivalents only:
    - Size of longest semantic equivalent
    - e.g. if (a xor b) can be re[...] xor instructions
  - For insertion of noise instru[...]
    - MSD (Maximum Standar[...]
    - Better than mean: mean [...]
    - Better than worst case

X : number of noise instructions to insert

P[X=0] = 0,99

P[X=10] = 0,01

Mean: 0,1 noise instructions inserted at every call

Worst case: 10 noise instructions inserted at every call
MSD: 0,9

Allocating size = MSD*number call gives a size 11 times shorter than using worst case!

In practice, MSD metric works well

STATIC ALLOCATION OF A REALISTIC SIZE

- **How to determine a realistic size for allocation?**
  - Worst case is terrible and never happens in programs long enough

    → need for a better metric
  - Worst case used for semantic equivalents only:
    - Size of longest semantic equivalent
    - e.g. if (a xor b) can be replaced by (a xor r) xor (b xor r), we reserve space for 3 xor instructions
  - For insertio[...]
    - MSD ([...]n) is used
    - Better [...]ams
    - Better [...]

Allocated size =

size of original instructions

+ worst case size of semantic equivalents

+ MSD * number of calls to noise instructions generator (usually equal to number of original instructions)

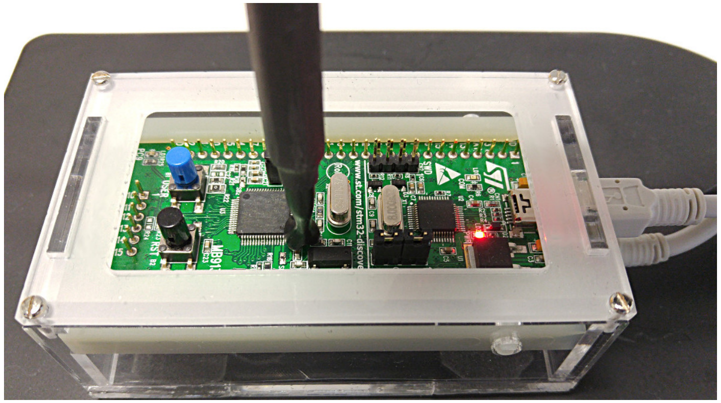
OVERFLOW PREVENTION

- **Statically compute size of useful instructions**
  - Knowledge of size of what comes next

- **Information is given to the generator**

- **Throughout generation: generator computes the size to keep for useful instructions**
  - Noise instruction insertion limited if necessary

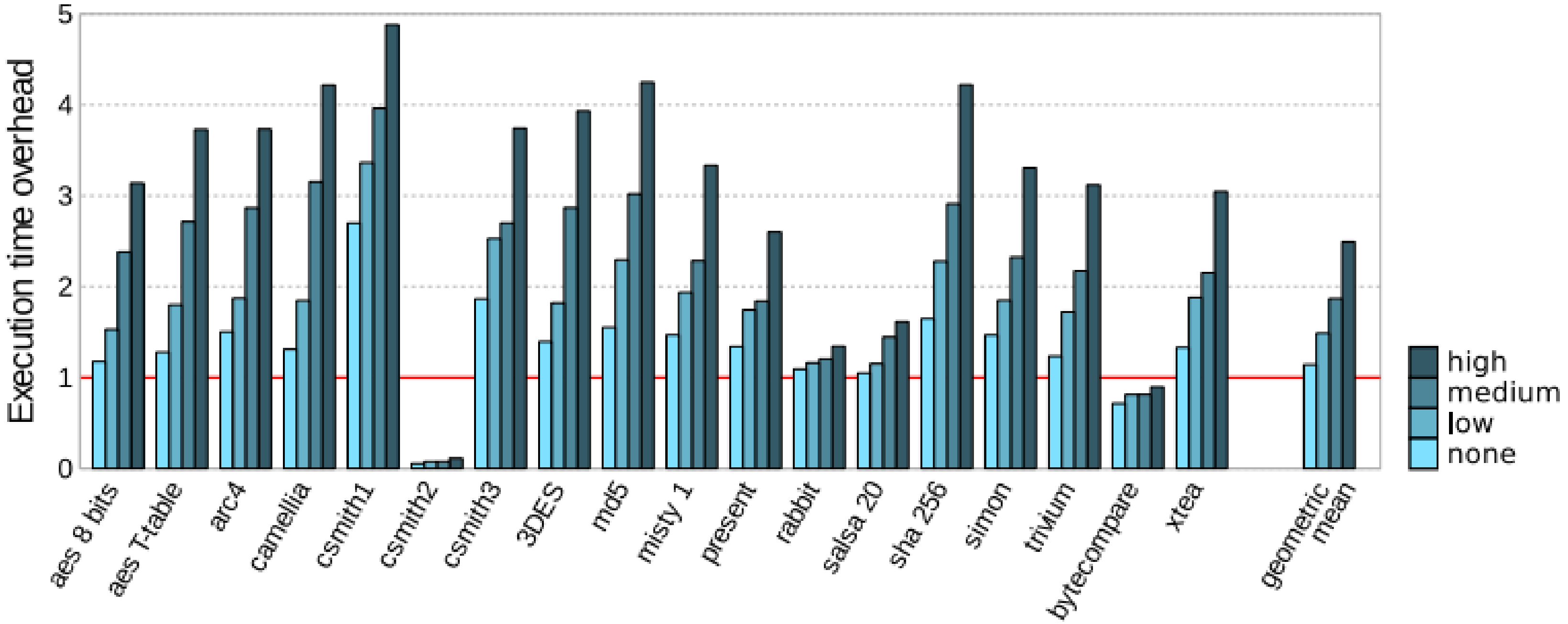
RESULTS



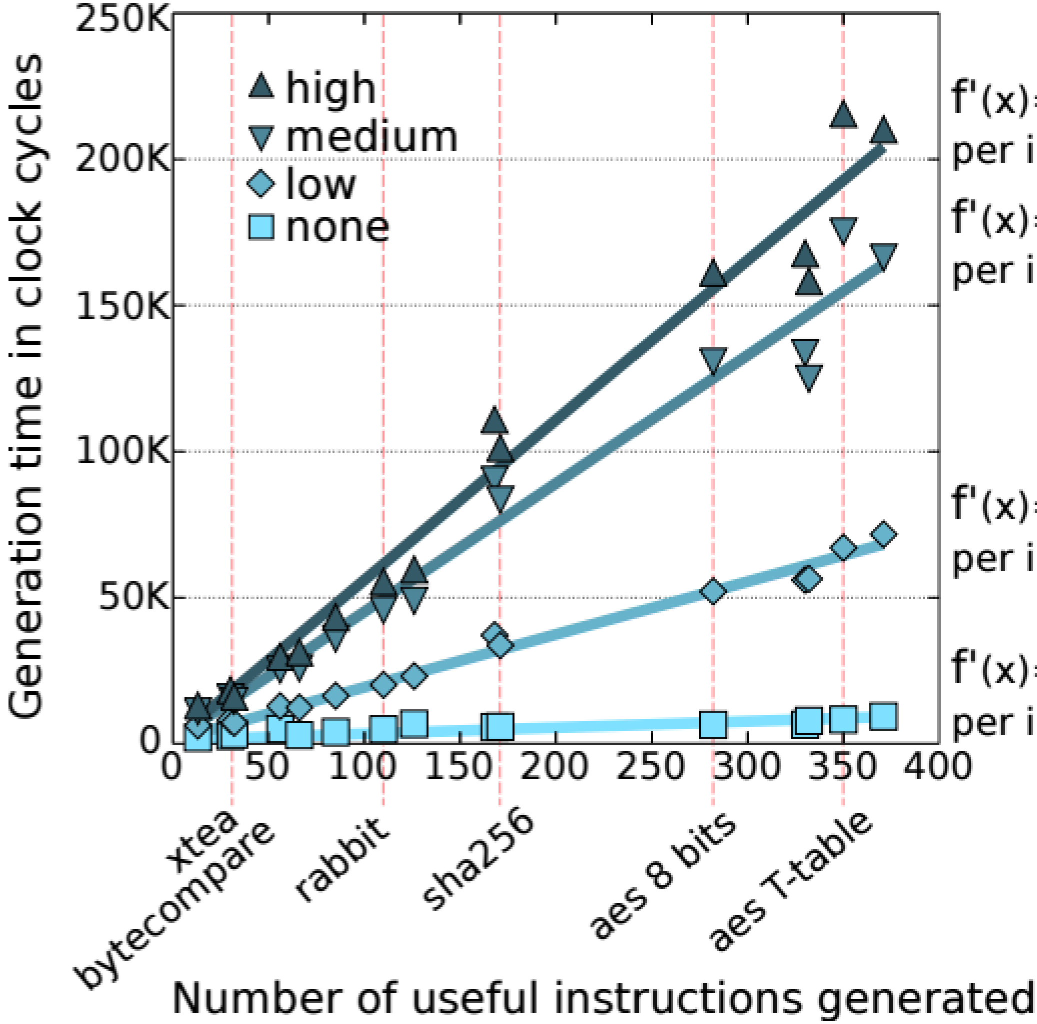
- **Performance evaluation**
  - 18 different test cases
  - Among them, 3 randomly generated tests
  - 4 different configurations
    - None: no polymorphism
    - Low: only noise instructions, generation is done every 250 executions
    - Medium: all transformations activated, generation is done every execution
    - High: all transformations activated, different probability model for noise instructions insertion, generation is done every execution
  - STM32 board (ARM cortex M3 – 24 MHz – 8kB of RAM)

- **Security evaluation**
  - Same as performance evaluation +
  - PicoScope 2208A, EM probe RF-U 5-2 (Langer), PA 303 preamplifier (Langer)
  - Sampling at 500 Msample/s with 8bits resolution, 24500 samples per trace

RESULTS: COMPARISON OF EXECUTION TIME OVERHEAD FOR 4 CONFIGURATIONS



RESULTS: COMPARISON OF COST OF GENERATION FOR 4 CONFIGURATIONS



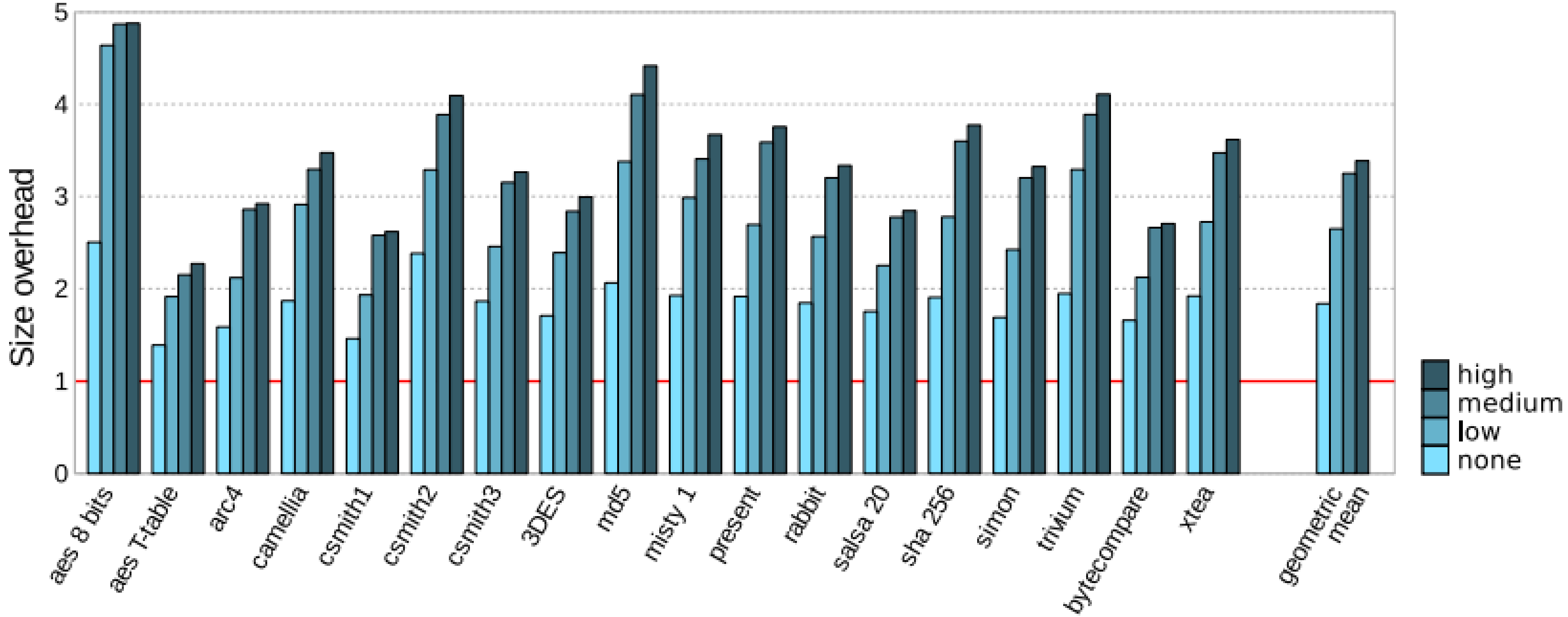
f'(x)=547cycles per instruction

f'(x)=439cycles per instruction

f'(x)=181cycles per instruction

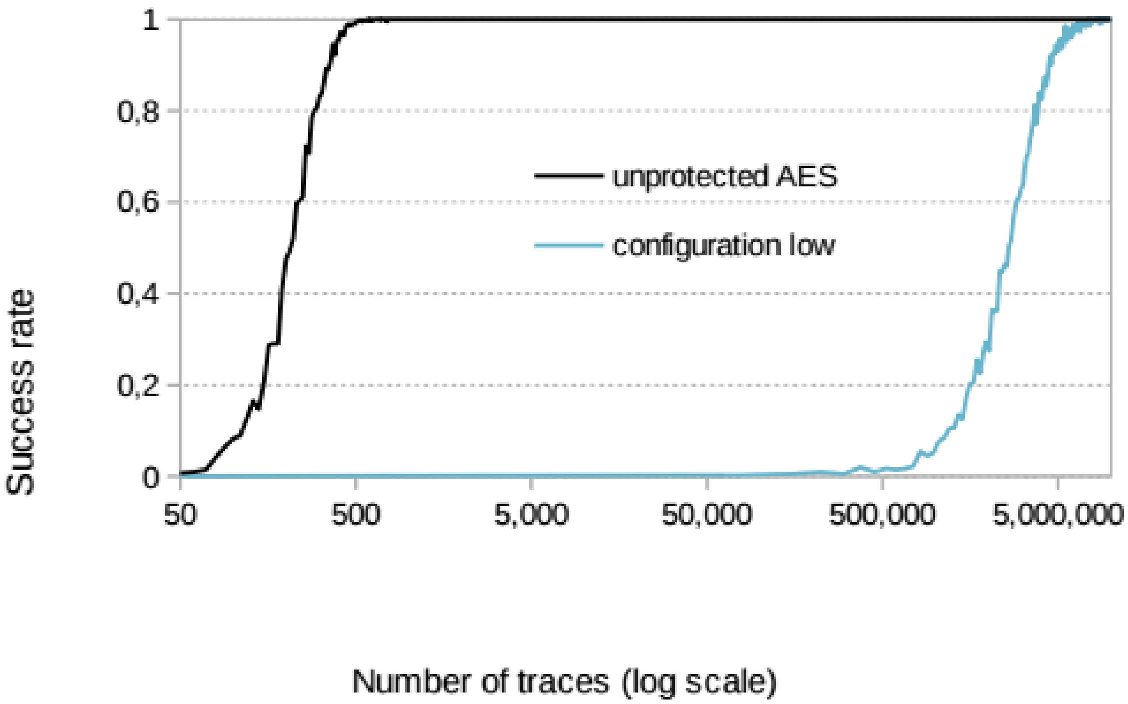
f'(x)=21cycles per instruction

RESULTS: COMPARISON OF CODE SIZE OVERHEAD FOR 4 CONFIGURATIONS



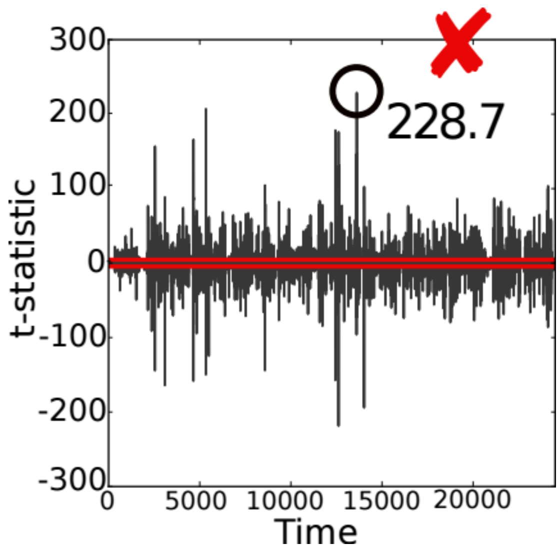
RESULTS: CPA FOR REFERENCE AND LOW

- **Attack on Sbox output with HW**
- **Srate at 0.8 in**
  - 290 traces for unprotected AES
  - 3 800 000 traces for configuration low
    - 13000 time more traces needed!
    - Execution time overhead of 2.8, including generation cost

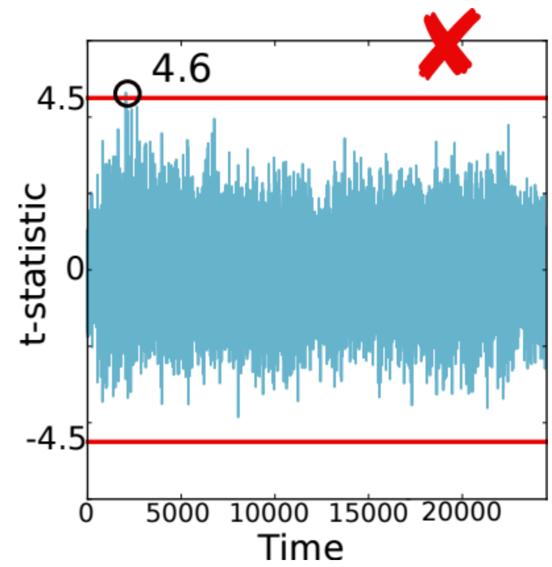


RESULTS: TTEST FOR 4 CONFIGURATIONS

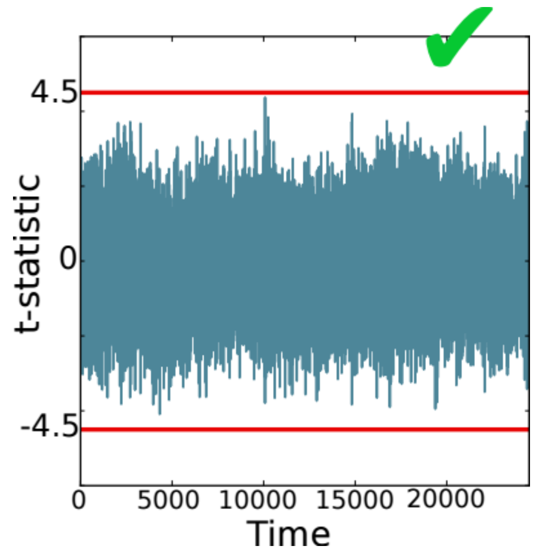
Reference



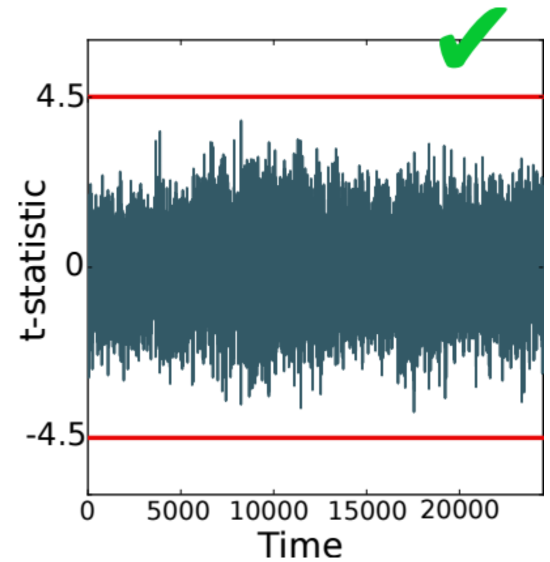
Low



Medium



High



CONCLUSION

- **Automatic AND configurable approach**
  - Works on any code
  - Allows to tune the trade off between performance and security
- **Specialization of generators**
  - Management of memory permission
  - Efficient code generation
- **Static allocation of realistic size + buffer overflow prevention**
- **Perspective: study the impact of polymorphism on the difficulty of mounting fault injection attack**

Commissariat à l'énergie atomique et aux énergies alternatives
17 rue des Martyrs | 38054 Grenoble Cedex
www.cea-tech.fr

Établissement public à caractère industriel et commercial | RCS Paris B 775 685 019