


The top section of the slide features a background of falling binary digits (0s and 1s) in a light gray color. On the left side, there is a red rectangular box containing the CEA Tech logo. The logo consists of the text "FROM RESEARCH TO INDUSTRY" in small, white, uppercase letters at the top, followed by "cea tech" in a larger, white, lowercase font. A thin green horizontal line is positioned below the text.

FROM RESEARCH TO INDUSTRY

cea tech

SOFTWARE COUNTERMEASURES AGAINST PHYSICAL ATTACKS IN EMBEDDED SYSTEMS

The bottom section of the slide has a background pattern of a circuit board, with various lines, nodes, and components in a light gray color.

Damien Couroussé, Nicolas Belleville | LIST / DACLE
Workshop PROSECCO | 20181106 | LIP6



leti Grenoble



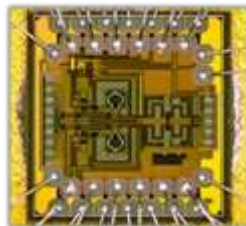
list Saclay



Digital design



Programming



Analog & MEMS



Signal processing



Imaging



Test

Embedded software and compiler

Automatic application of HW/SW
counter-measures against physical attacks



Low-power and FDSOI design

Low-power crypto-accelerators protected
against physical attacks



Attack monitors and detection by machine
learning against combined attacks



Design in emerging technologies

Secure virtual-machines in cloud



Secure, programmable in-memory computing

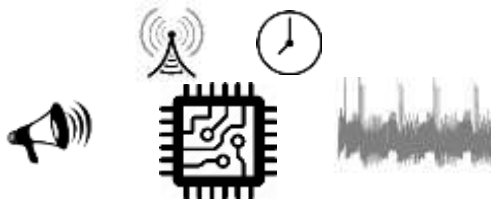


Design for security in new memory
technologies

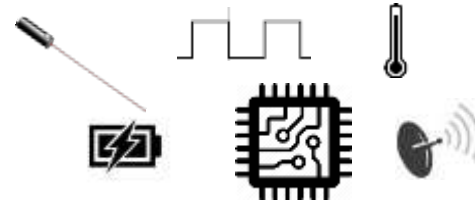


- One of the major threats against secure embedded systems
 - The only effective classes of attacks against crypto-systems
 - Relevant in many cases against cyber-physical systems: bootloaders, firmware upgrade, etc.

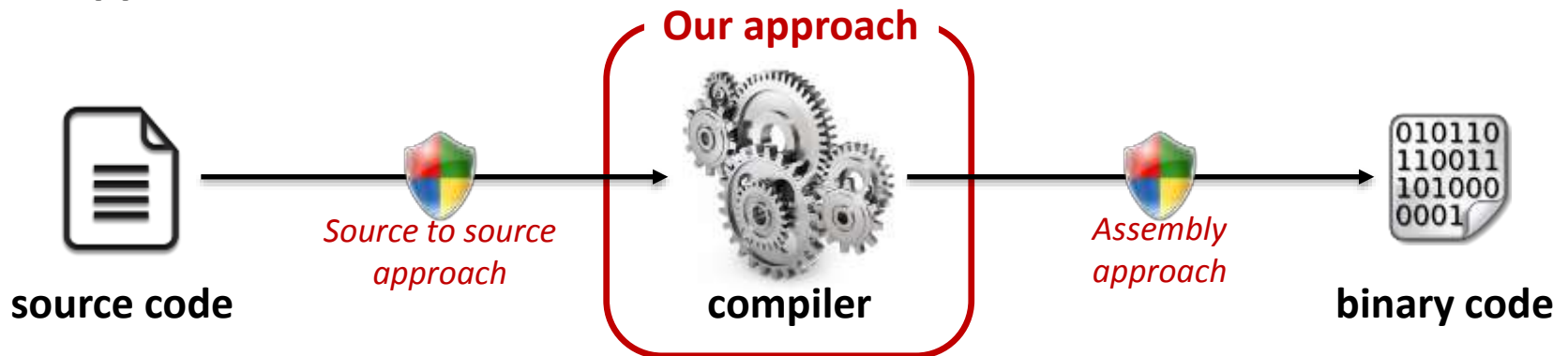
side channel attacks



fault attacks



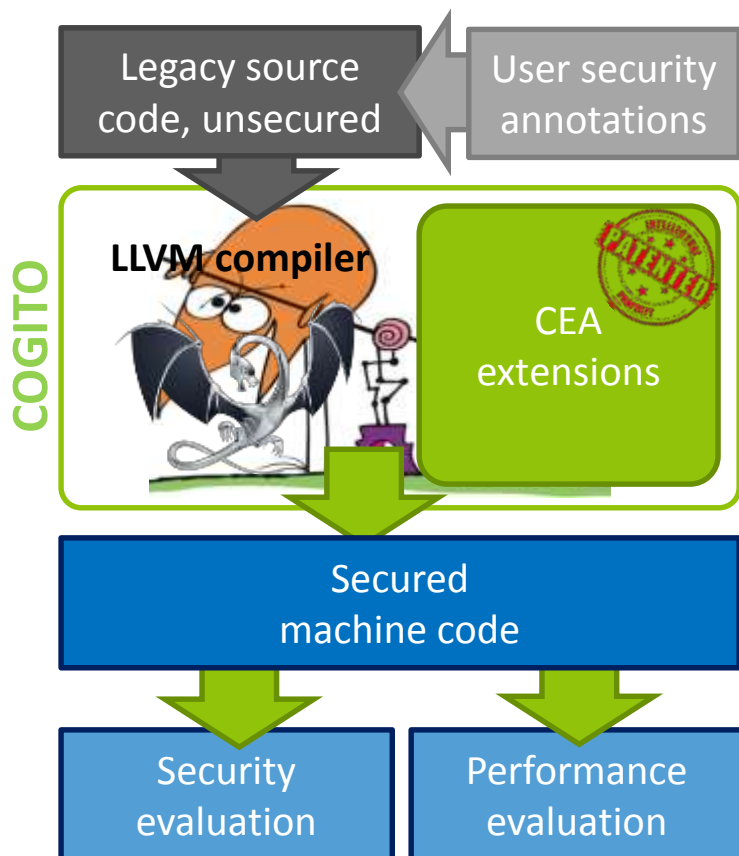
- Application of software countermeasures in a software toolchain



➔ Towards **secured** and **efficient** software components

Automated application of software countermeasures against physical attacks

➔ A toolchain for the compilation of secured programs



- Several countermeasures

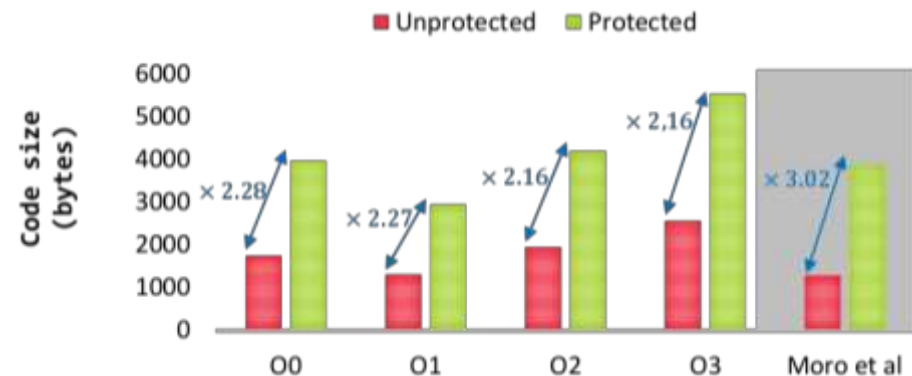
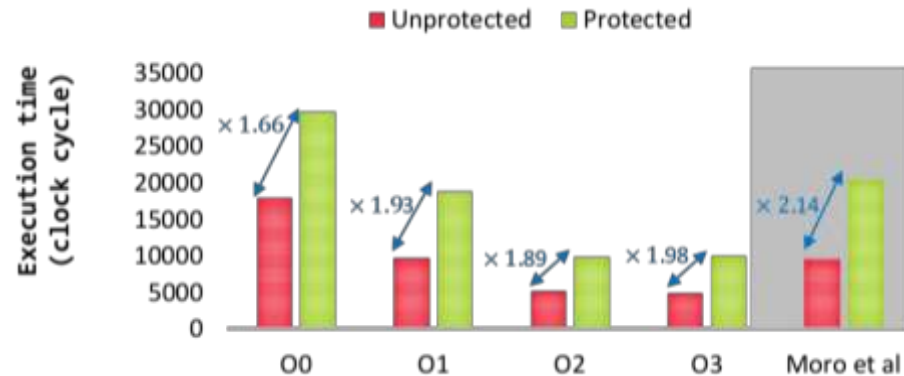
- **Fault tolerance**, including multiple fault injections
- **Fault detection & Control-Flow Integrity**
 - Combined with integrity of execution paths at the granularity of a single machine instruction
- **Side channel hiding**

- Tools for **security and performance evaluations**

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

Aim: the component is not perturbed by the injection of multiple faults

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



- Fine-grained application** of the countermeasure **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.

Objectives: detection of a fault injection.

Combined protections:

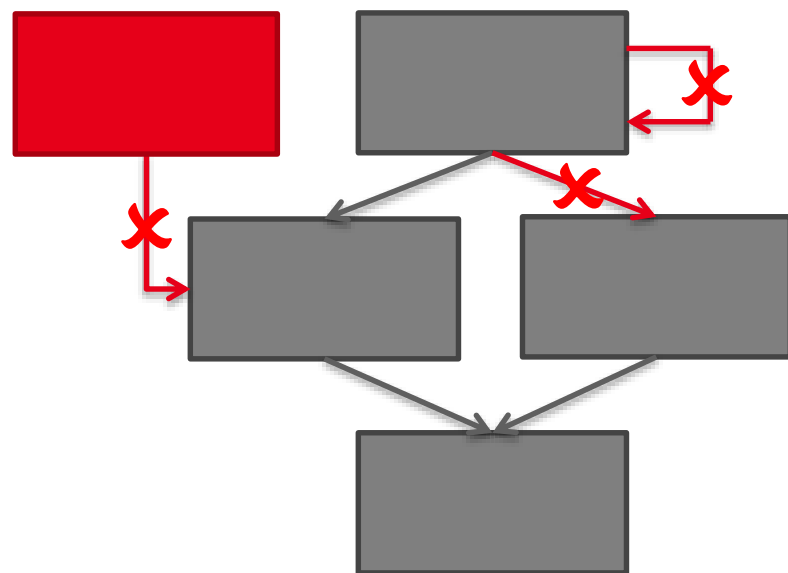
- Protection of the control-flow of an application (Control-Flow Integrity)
- Beyond CFI: protection of branchless sequences of instructions, at the granularity of a single machine instruction

Coverage

- Alteration of the PC (instruction skips, branches)
- Corruption of branches
- Alteration of branch conditions

Implementations

- Software only countermeasure. Implementation for ARM
- HW-SW countermeasure. Fine-grain execution integrity, verification & authentication. WIP implementation for RISC-V



Secured transition



Secured application component

Illegal transition



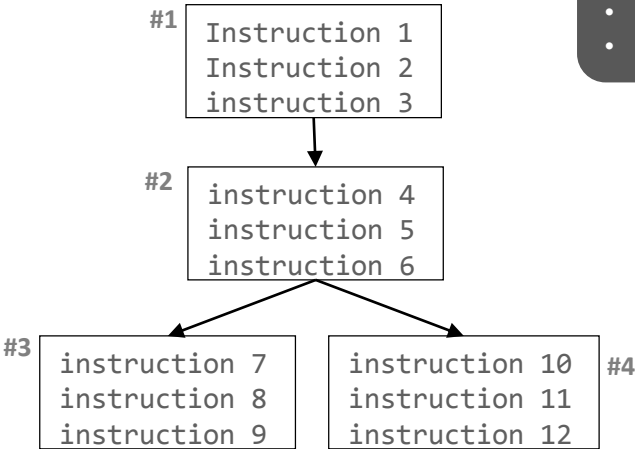
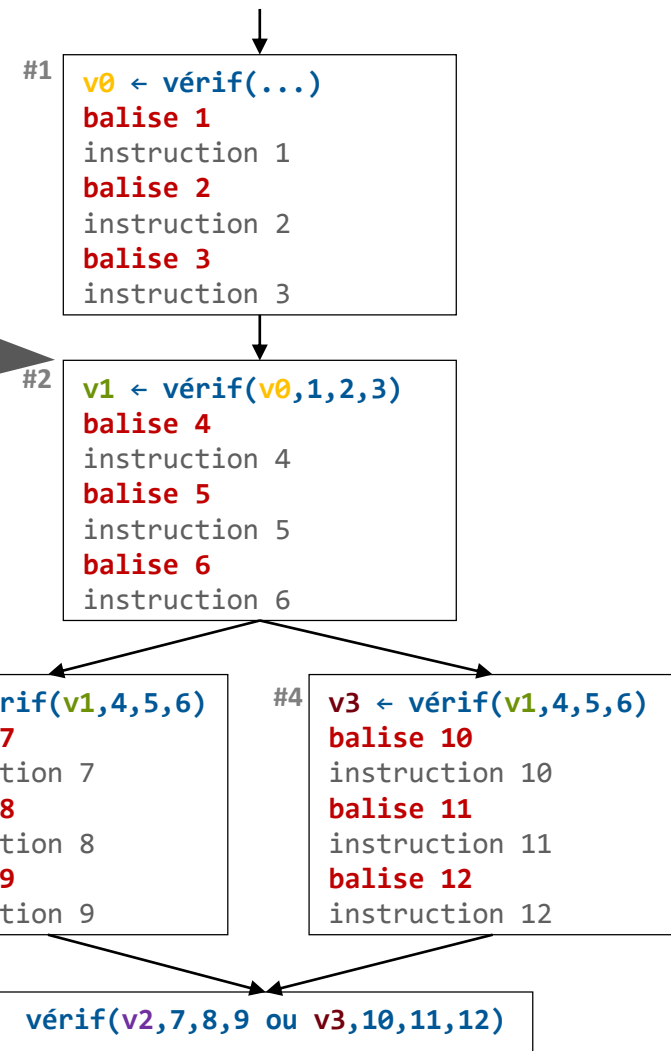
Unsecured component



PRINCIPE DE FONCTIONNEMENT

On vérifie le chemin d'exécution pour le bloc de base #1 :

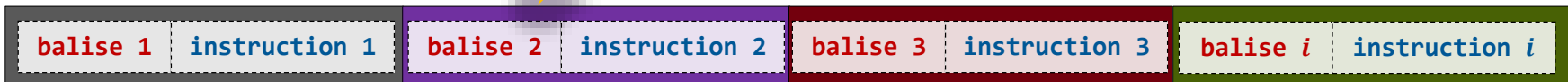
- La vérification précédente **v0** est correcte
- L'exécution a rencontré les balises 1, 2 et 3



Flot de contrôle initial

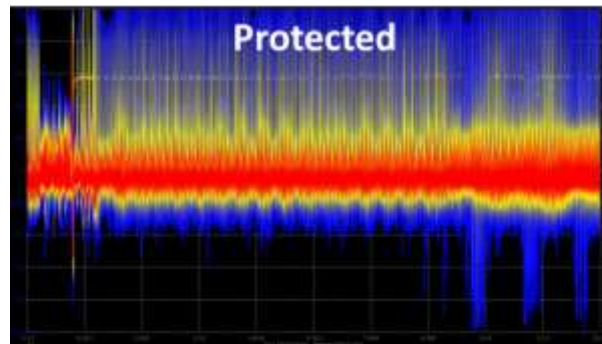
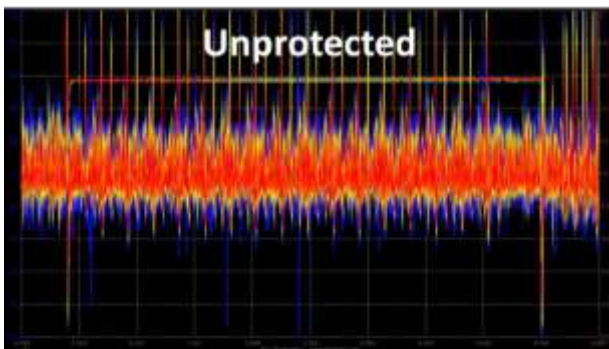
Alignement des instructions de telle sorte que chacune soit inséparable de sa balise correspondante dans une fenêtre de tire

Flot de contrôle sécurisé contre le détournement de flot de contrôle et contre l'altération d'instruction

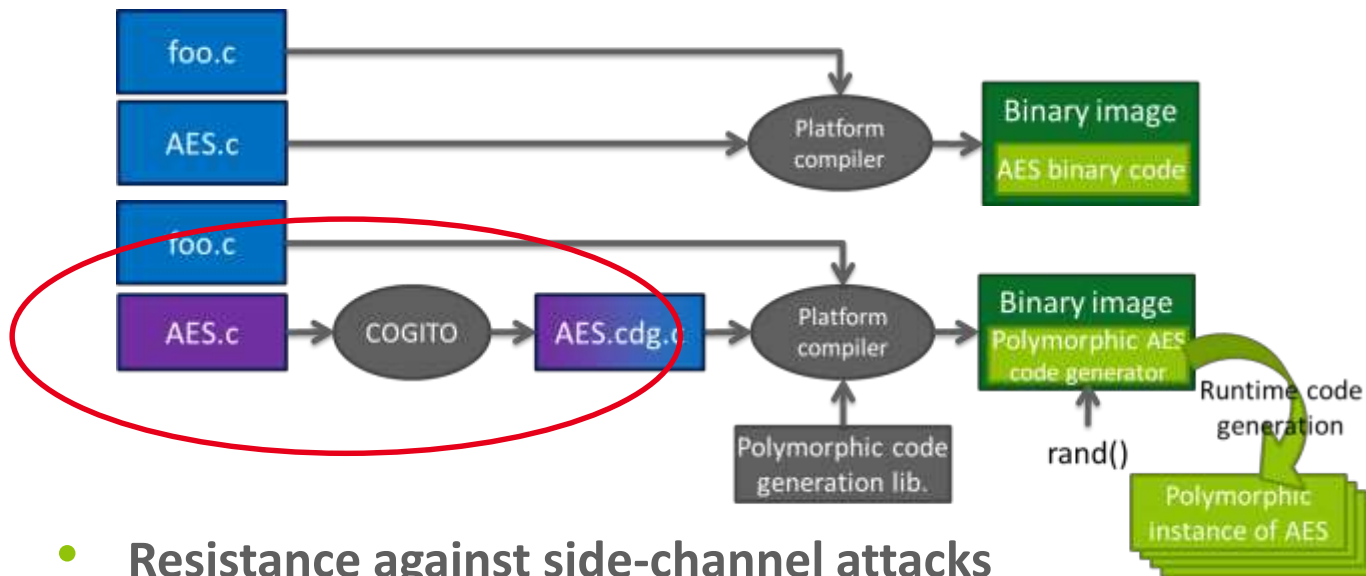


Code polymorphism: regularly changing the behavior of a (secured) component, at runtime, while maintaining unchanged its functional properties,

- **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
- Implementation with **runtime code generation**
- **Demonstrator with 8KBytes RAM – STM32**



- Toolchain for the application of code polymorphism, targetting embedded systems



- Resistance against side-channel attacks
 - CPA: 50 traces \rightarrow 5.10^6 traces
 - TLVA evaluation / t-test
 - Increased resistance to resynchronisation of observation traces
 - Increased resistance against template attacks (study on verifyPIN)
- Compatible with certification standards – Common Criteria
- Compatible with program encryption

**AUTOMATED SOFTWARE PROTECTION
FOR THE MASSES
AGAINST SIDE-CHANNEL ATTACKS**

- **Ability to change the observable behaviour of a software component without changing its functional properties**
- **Hiding countermeasure**
 - does not remove the leakage
- **Our approach:**
 - use of runtime code generation to make the code vary
 - specialized generator: each polymorphic function has its own generator
 - set of assembly-level code transformation

PROBLEMS WE WANTED TO ANSWER

- How to write automatically a generator?
- Runtime code generation is usually expensive
- Runtime code generation needs W and X permissions
- Code size varies from one generation to another...
 - Semantic equivalent
 - Insertion of noise instructions
- ...but platform may not have dynamic memory allocation features

- **How to generate a generator using a compiler?**
- **What transformations does the generator use to generate a different code at each generations?**
- **How to guarantee that the allocated buffer is never writable and executable at the same time?**
- **How to allocate a realistic size, and to prevent overflows during code generation?**
- **Experimental evaluation**

- **How to generate a generator using a compiler?**
- What transformations does the generator use to generate a different code at each generations?
- How to guarantee that the allocated buffer is never writable and executable at the same time?
- How to allocate a realistic size, and to prevent overflows during code generation?
- Experimental evaluation

AUTOMATIC APPLICATION OF CODE POLYMORPHISM

- 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

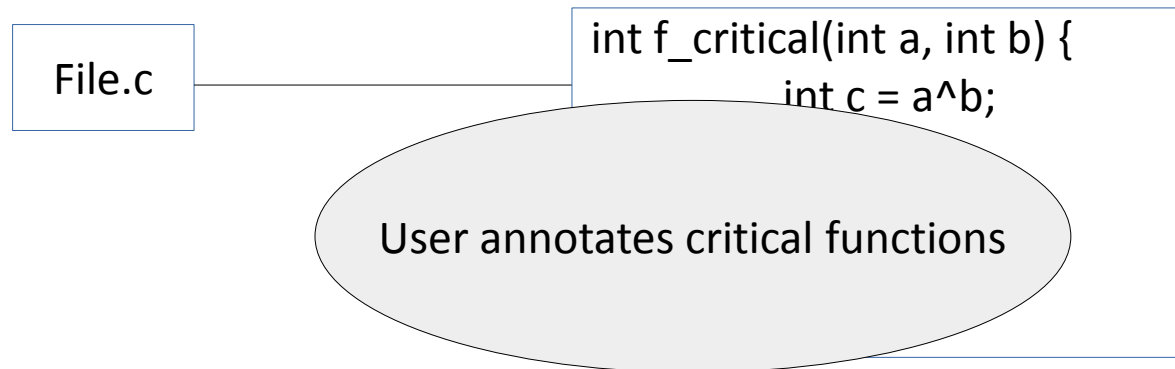
- Start from a C file
- Produce a new C file with polymorphism countermeasure applied to target functions

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

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



AUTOMATIC APPLICATION OF CODE POLYMORPHISM

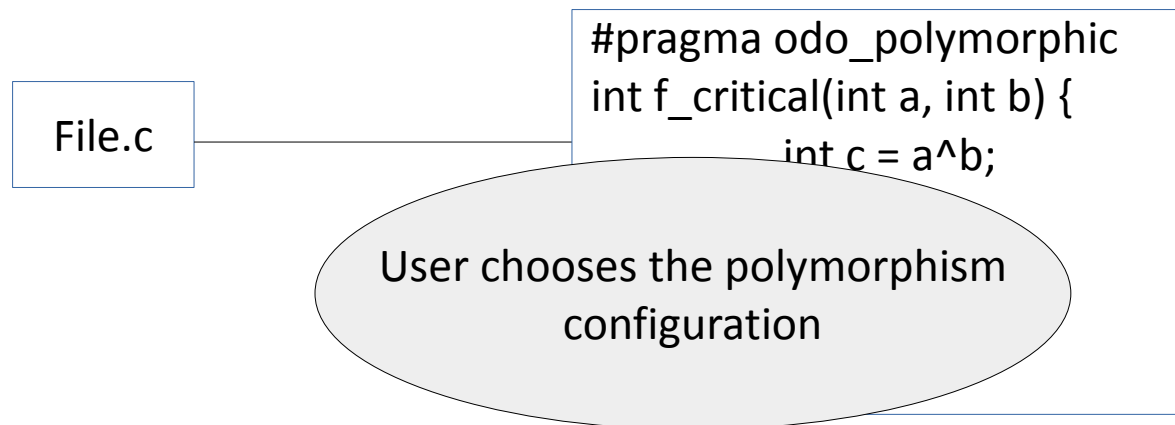
- Start from a C file
- Produce a new C file with polymorphism countermeasure applied to target functions

File.c

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

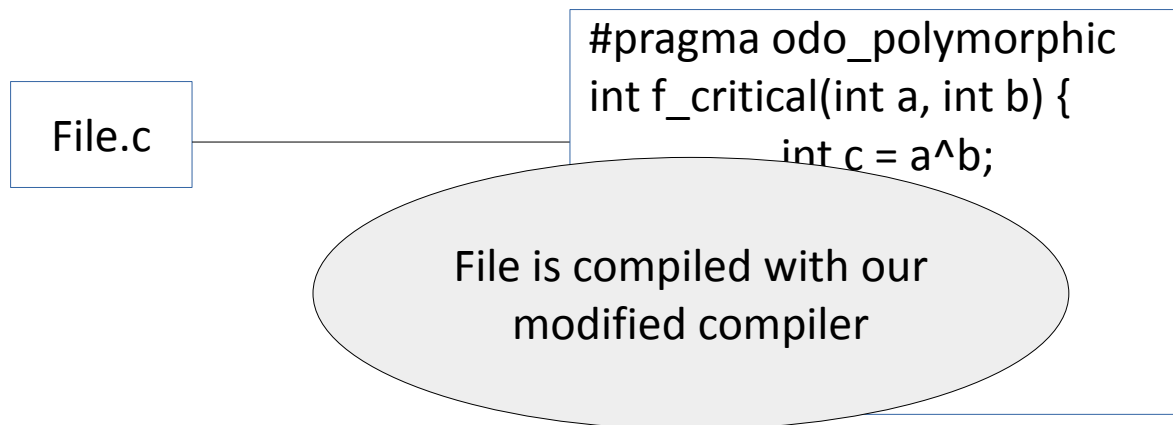
AUTOMATIC APPLICATION OF CODE POLYMORPHISM

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



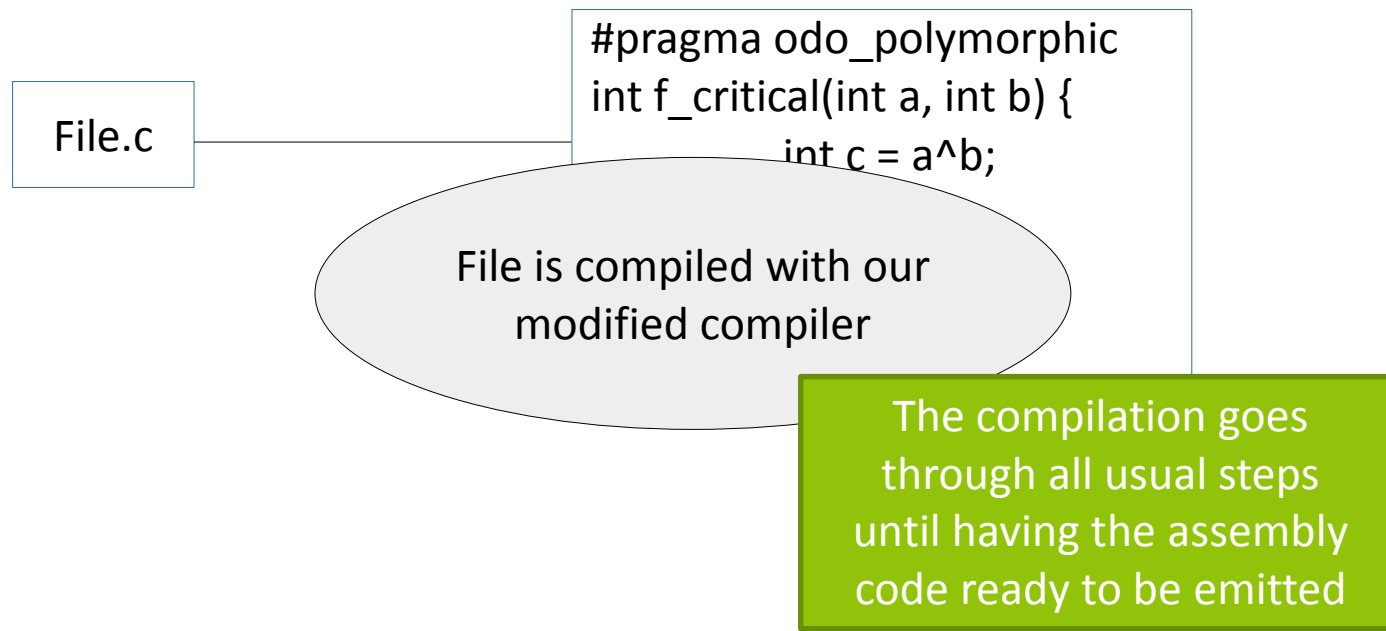
AUTOMATIC APPLICATION OF CODE POLYMORPHISM

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



AUTOMATIC APPLICATION OF CODE POLYMORPHISM

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



AUTOMATIC APPLICATION OF CODE POLYMORPHISM

- Start from a C file
- Produce a new C file with polymorphic 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

- Start from a C file
- Produce a new C file with polymorphic 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

- Start from a C file
- Produce a new C file with polymorphic 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

- Start from a C file
- Produce a new C file with polymorphic 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

- Start from a C file
- Produce a new C file with polymorphic 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);
}
```

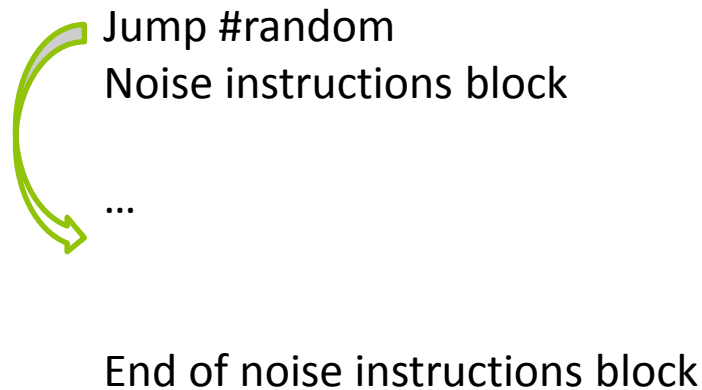
- How to generate a generator using a compiler?
- **What transformations does the generator use to generate a different code at each generations?**
- How to guarantee that the allocated buffer is never writable and executable at the same time?
- How to allocate a realistic size, and to prevent overflows during code generation?
- Experimental evaluation

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 instruction
 - Semantic equivalents available for a limited number of instructions
 - Ex: $a \text{ xor } b \iff (a \text{ xor } r) \text{ xor } (b \text{ xor } r)$
- **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

CODE TRANSFORMATIONS USED AT RUNTIME

- **Dynamic noise sequences inserted in between useful instructions**
 - Exactly like « classic » noise instructions
 - Sequence of noise instructions with a random jump



- Jump #random actually done in 6 assembly instructions
- Lowers correlation between generation phase and execution phase
- Allows use of wider regeneration period

- How to generate a generator using a compiler?
- What transformations does the generator use to generate a different code at each generations?
- How to guarantee that the allocated buffer is never writable and executable at the same time?
- How to allocate a realistic size, and to prevent overflows during code generation?
- Experimental evaluation

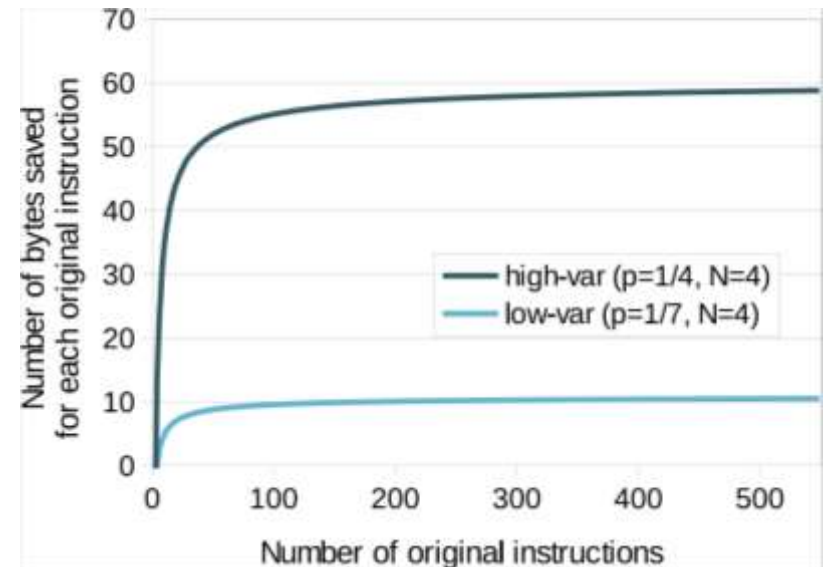
MANAGEMENT OF MEMORY PERMISSIONS

- **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

- How to generate a generator using a compiler?
- What transformations does the generator use to generate a different code at each generations?
- How to guarantee that the allocated buffer is never writable and executable at the same time?
- How to allocate a realistic size, and to prevent overflows during code generation?
- Experimental evaluation

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 $(a \text{ xor } b)$ can be replaced by $(a \text{ xor } r) \text{ xor } (b \text{ xor } r)$, we reserve space for 3 xor instructions
- **For insertion of noise instruction:**
 - threshold-based allocation: find allocation size such as probability of overflow is below a user defined threshold
 - Much better than worst case!
 - See results with threshold = 10^{-6}



- **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

- How to generate a generator using a compiler?
- What transformations does the generator use to generate a different code at each generations?
- How to guarantee that the allocated buffer is never writable and executable at the same time?
- How to allocate a realistic size, and to prevent overflows during code generation?
- **Experimental evaluation**

- **Performance evaluation**

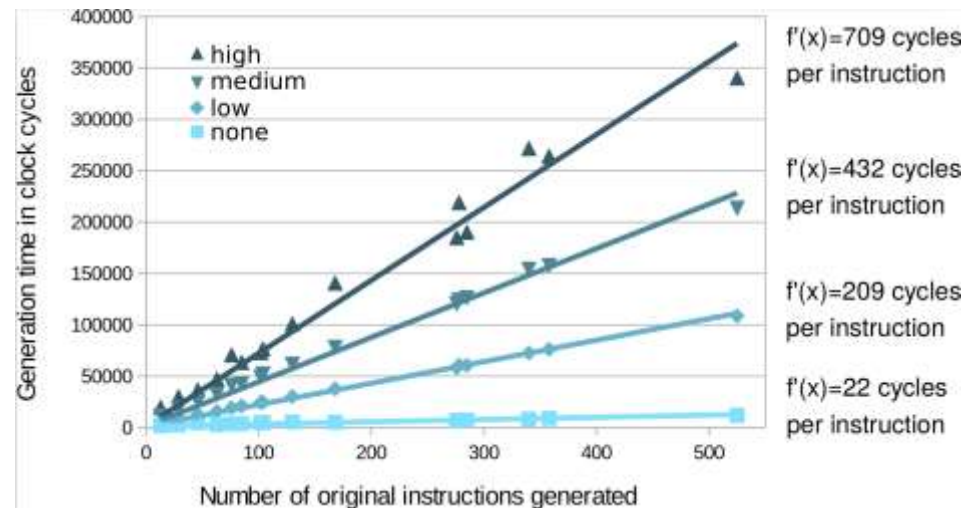
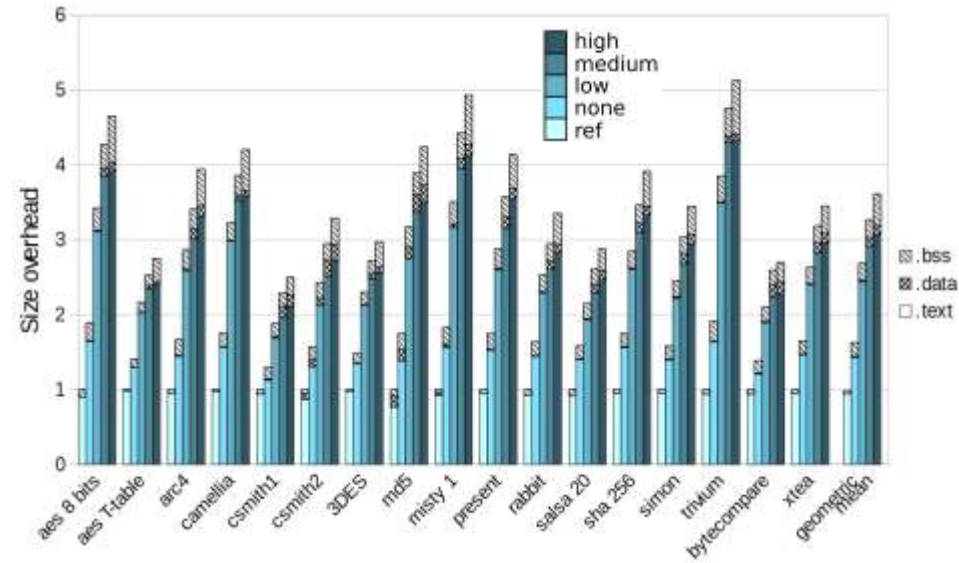
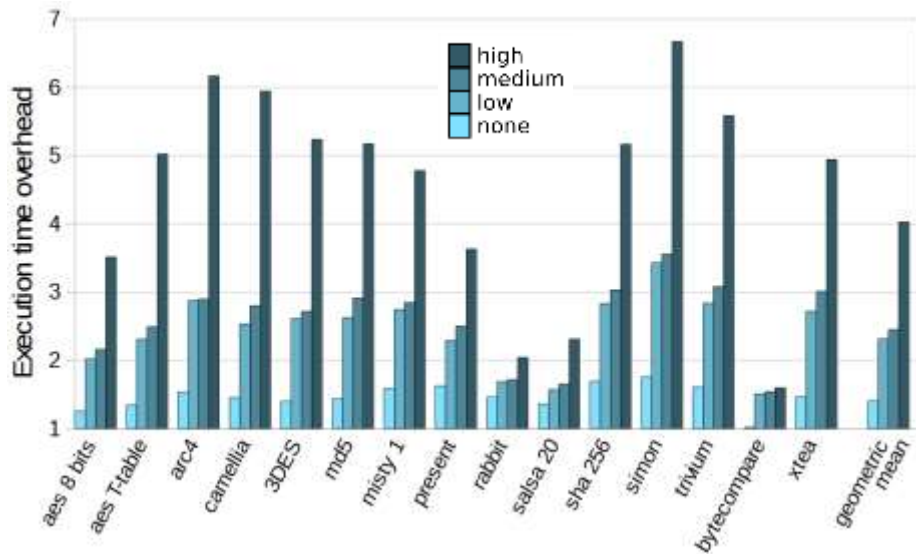
- 15 different test cases
- 4 different configurations
 - None: no polymorphism
 - Low: only noise instructions, generation is done every 250 executions
 - Medium: all transformations activated, generation is done every execution
 - High: all transformations activated, different probability model for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



- **Security evaluation**

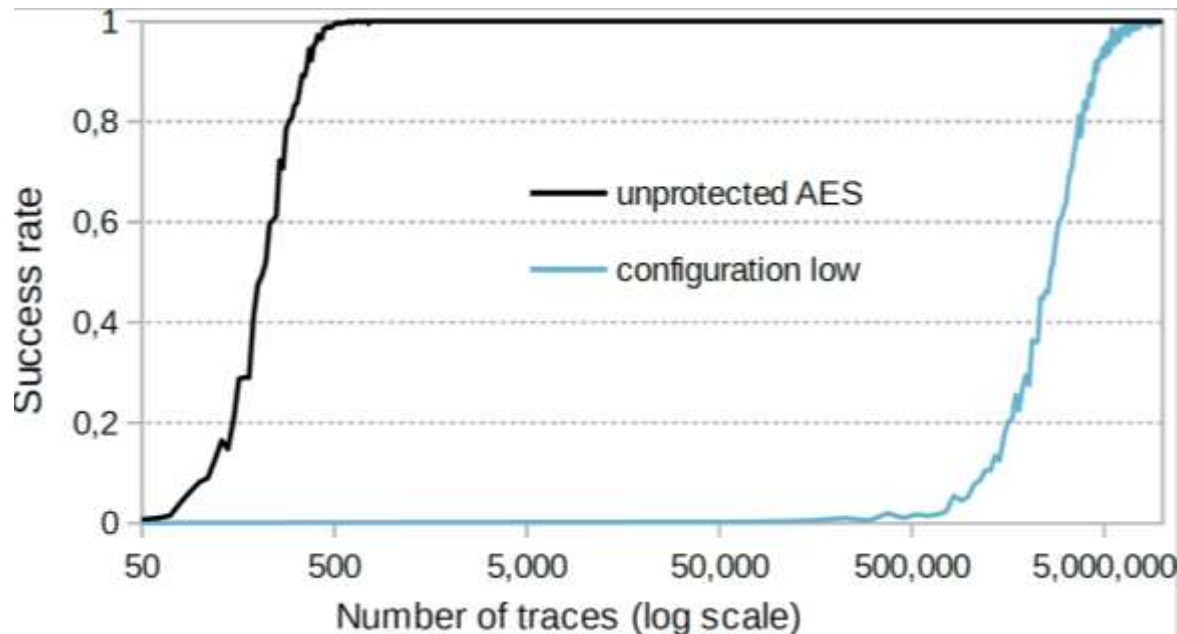
- Same as performance evaluation +
- PicoScope 2208A, EM probe RF-U 5-2 (Langer), PA 303 preamplifier (Langer)
- Sampling at 500 Msample/s with 8bits resolution, 24500 samples per trace

RESULTS: PERFORMANCE



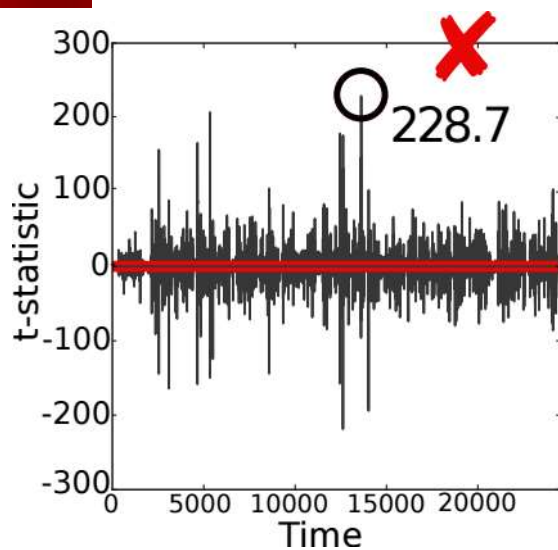
RESULTS: CPA FOR REFERENCE AND LOW

- **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 2.5, including generation cost

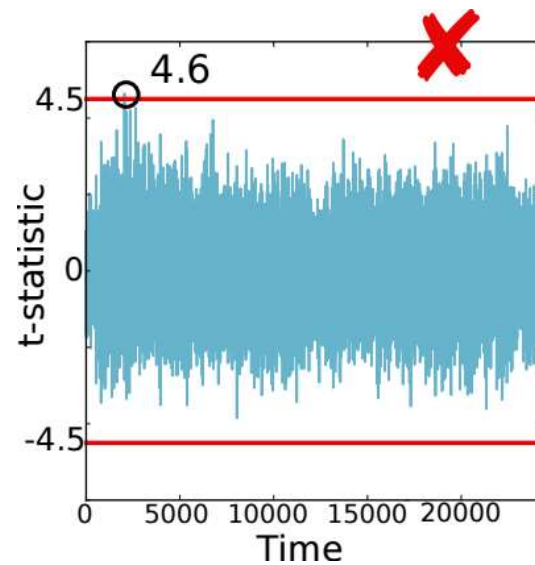


RESULTS: TTEST FOR 4 CONFIGURATIONS

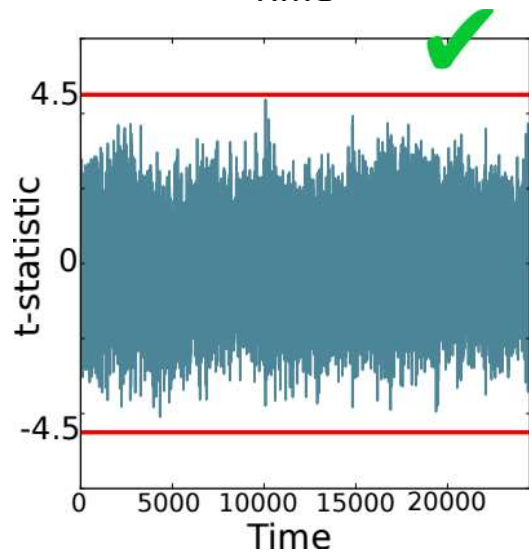
Reference



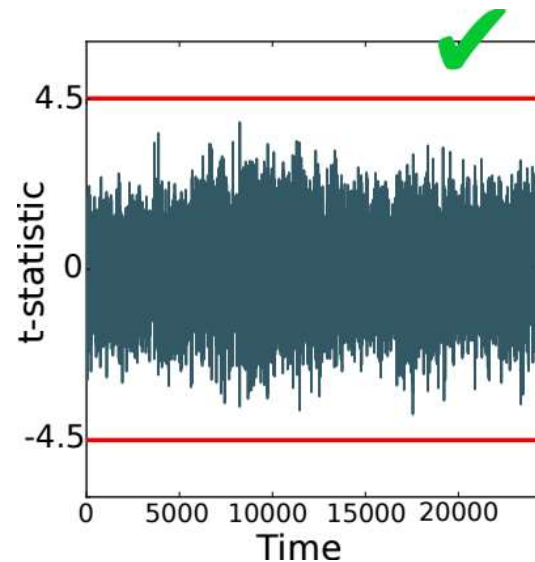
Low



Medium



High



- **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**
- **Future work: combination with other countermeasures**