FROM RESEARCH TO INDUSTRY

cea tech

Compilation for cyber-security in embedded systems

Workshop SERTIF

Damien Couroussé
CEA – LIST / LIALP; Grenoble Université Alpes
damien.courousse@cea.fr

PHYSICAL ATTACKS

Typology of attacks

- **Cryptanalysis**

  Out of our scope
- **Passive attacks. side channel attacks**

  Observations: power, electro-magnetic, execution time, temperature, etc.
- **Active attacks. fault attacks**

  Over/under voltage, laser, ion beam, EM, clock glitches…
- **Reverse engineering**

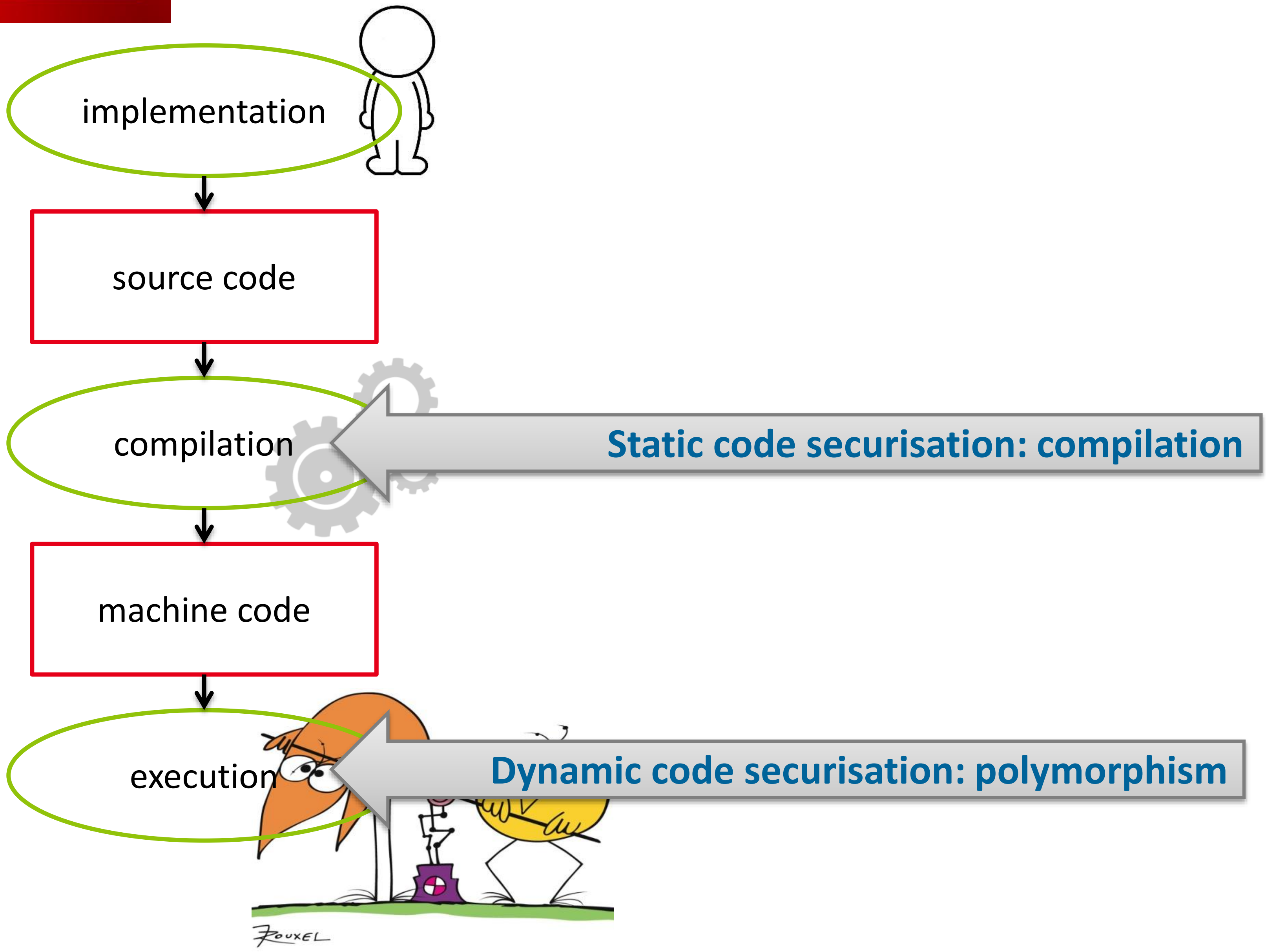
  Hardware inspection: mechanical or chemical etching, scope observation…
  Software inspection: debug, memory dumps, code analysis…
  Using physical attacks: SCARE, FIRE…
- **Logical attacks**

  Not (yet) considered

Real world attacks are different from research literature

- **First step. Global analysis, calibration of the attack bench(es), identification of weaknesses**
- **Second step. The textbook attack: a focused attack on a known weakness**

FROM THE IDEA TO A PRODUCT: A DETAILED PLAN



implementation

source code

compilation

machine code

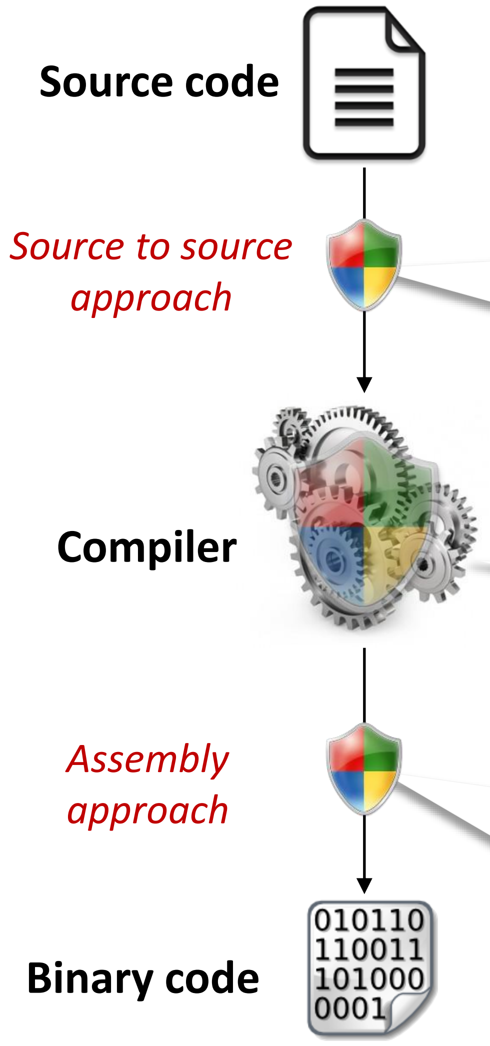
execution

Static code securisation: compilation

Dynamic code securisation: polymorphism

STATIC COMPILATION OF PROTECTIONS AGAINST FAULT ATTACKS

AUTOMATED APPLICATION OF COUNTERMEASURES WITH STATIC COMPILATION



Source code

Source to source approach

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

Compiler

← our approach

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

Assembly approach

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

Binary code

COMPILATION OF A COUNTERMEASURE AGAINST INSTRUCTION SKIP FAULT ATTACKS

- Attack model: faults, instruction skip
- Protection model: instruction redundancy
  - Formally verified countermeasure model [Moro et al., 2014]

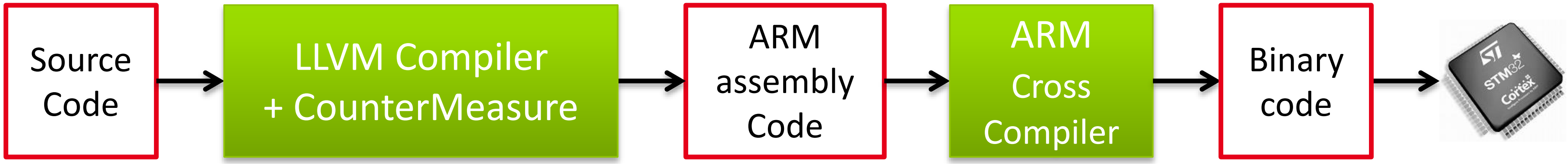
- **Platform**

  STM32 F100: ARM Cortex-M3

  Frequency: 24 MHZ

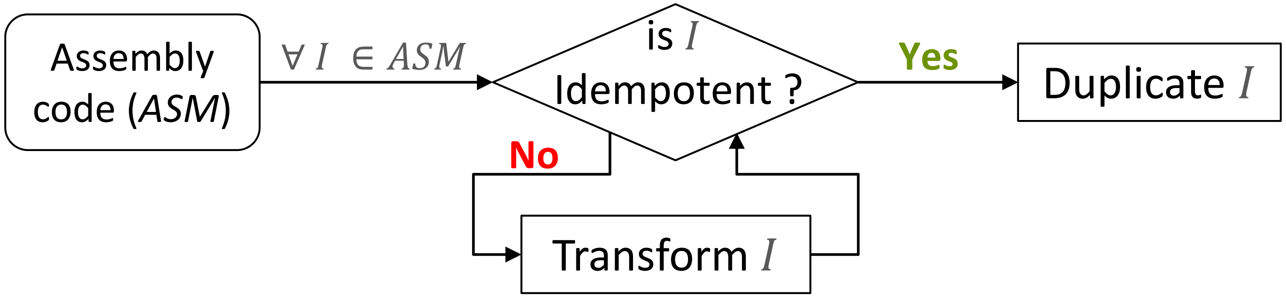
  Instruction Set: Thumb2

- Workflow



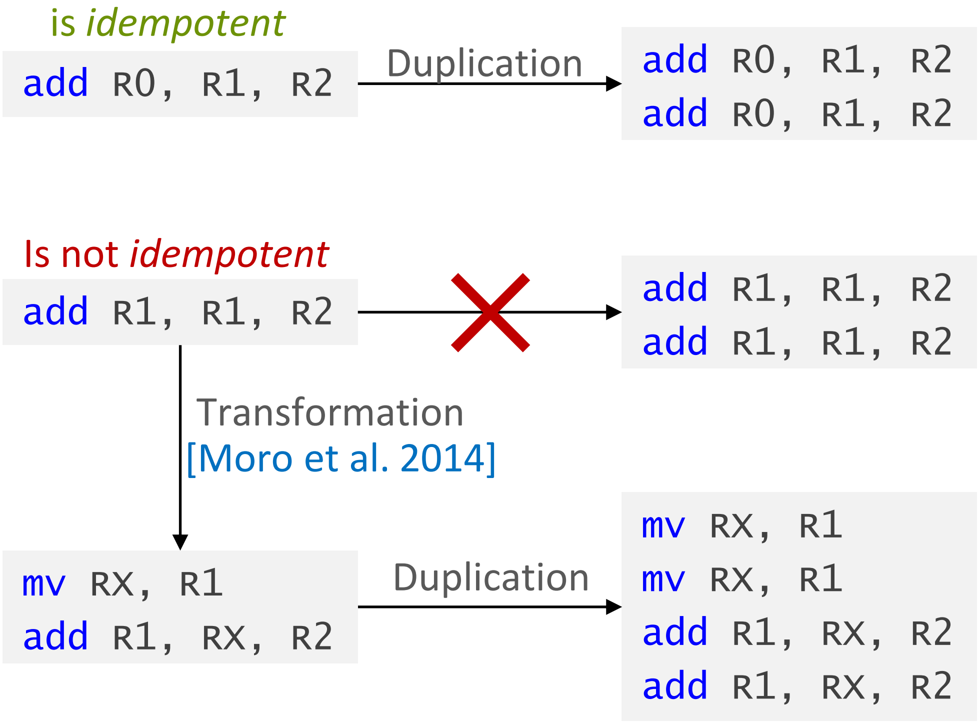
Source Code → LLVM Compiler + CounterMeasure → ARM assembly Code → ARM Cross Compiler → Binary code →

INSTRUCTION DUPLICATION SCHEME



> *"An instruction is idempotent when it can be* ***re-executed*** *several times with always the same result"*

Example



is *idempotent*

```
add R0, R1, R2
```

Duplication

```
add R0, R1, R2
add R0, R1, R2
```

Is not *idempotent*

```
add R1, R1, R2
```

```
add R1, R1, R2
add R1, R1, R2
```

Transformation
[Moro et al. 2014]

```
mv RX, R1
add R1, RX, R2
```

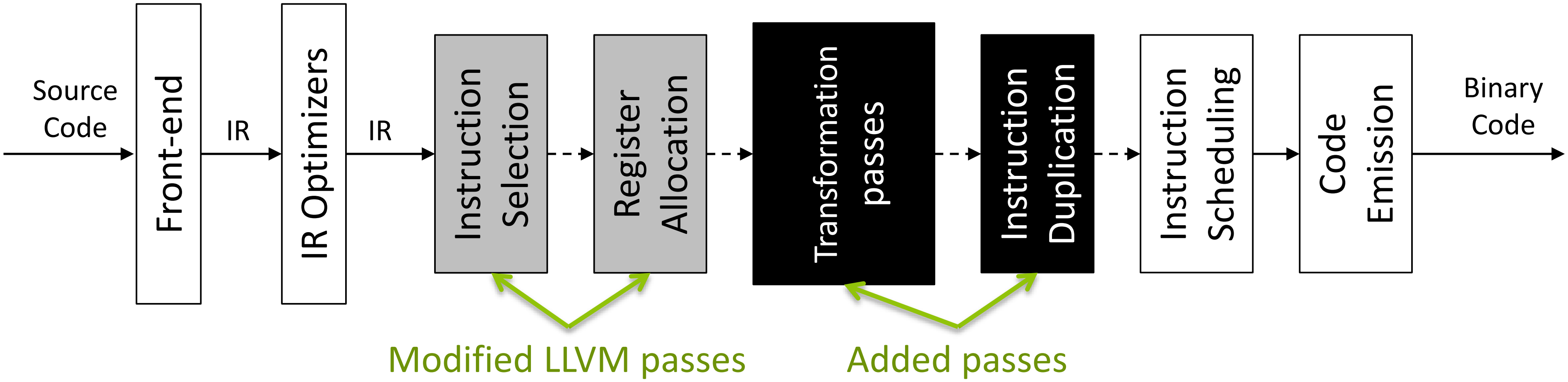
Duplication

```
mv RX, R1
mv RX, R1
add R1, RX, R2
add R1, RX, R2
```

Issues

- **How to find a free register at the assembly code level ?**
  - [Barenghi et al. 2010]: ad hoc implementation. the number of free registers is known for their implemented AES
  - [Moro et al. 2014]: use of the scratch register `r12`
- **Overhead:**
  - At least × **4** for each instruction
  - [Moro et al. 2014] Reported × **14** for the ARM instruction: `umlal`

INSTRUCTION DUPLICATION WITH LLVM



- Instruction selection
  - Force three-operands instructions
- Register allocation
  - Force the use of different registers for source and destination operands

```
add r0, r0, r1    =>    add r2, r0, r1
```

- Transformation passes
  - Transformation of non-idempotent instructions into a sequence of idempotent ones
- Instruction duplication
  - Straightforward. Could (should?) be executed *after* instruction scheduling

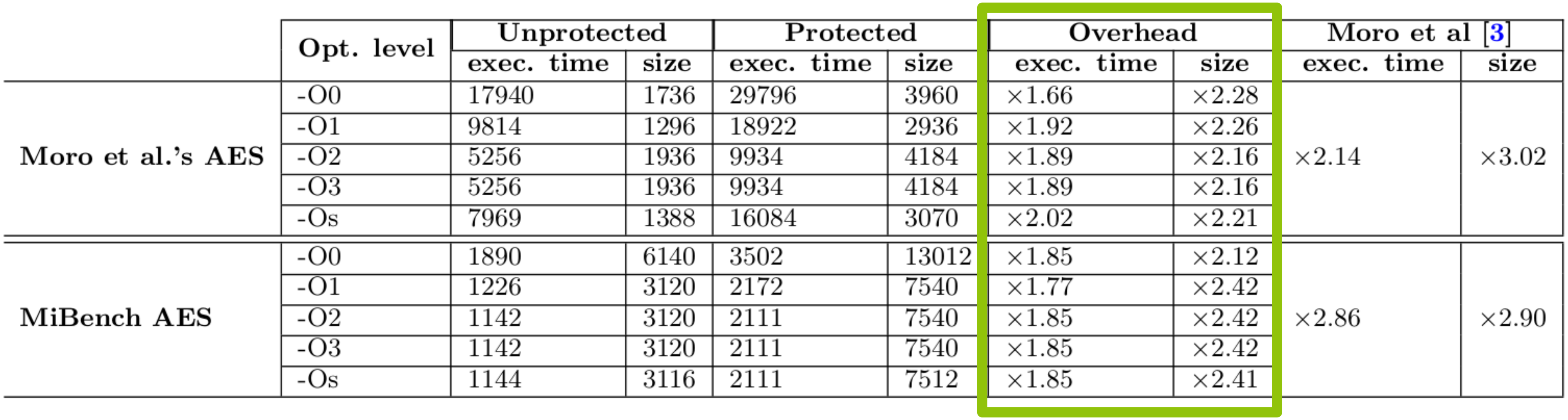
COMPILATION OF A COUNTERMEASURE AGAINST INSTRUCTION SKIP FAULT ATTACKS

- Attack model: faults, instruction skip
- Protection model: instruction redundancy
  - Formally verified countermeasure model [Moro et al., 2014]

- **Platform**

  STM32 F100: ARM Cortex-M3
  Frequency: 24 MHZ
  Instruction Set: Thumb2

- Experimental results for AES



	Opt. level	Unprotected		Protected		Overhead		Moro et al [3]	
		exec. time	size	exec. time	size	exec. time	size	exec. time	size
Moro et al.'s AES	-O0	17940	1736	29796	3960	×1.66	×2.28	×2.14	×3.02
	-O1	9814	1296	18922	2936	×1.92	×2.26		
	-O2	5256	1936	9934	4184	×1.89	×2.16		
	-O3	5256	1936	9934	4184	×1.89	×2.16		
	-Os	7969	1388	16084	3070	×2.02	×2.21		
MiBench AES	-O0	1890	6140	3502	13012	×1.85	×2.12	×2.86	×2.90
	-O1	1226	3120	2172	7540	×1.77	×2.42		
	-O2	1142	3120	2111	7540	×1.85	×2.42		
	-O3	1142	3120	2111	7540	×1.85	×2.42		
	-Os	1144	3116	2111	7512	×1.85	×2.41		

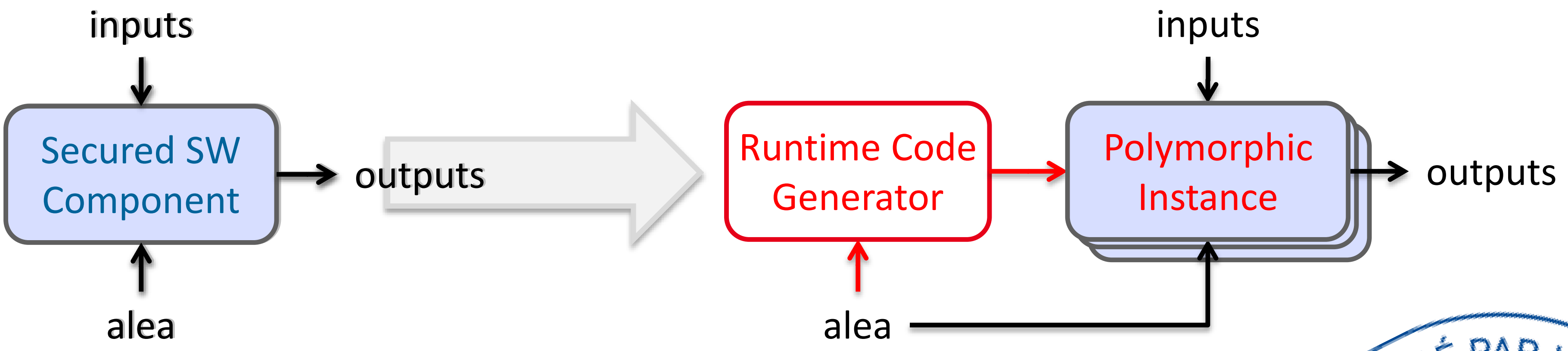
T. Barry, D. Couroussé, and B. Robisson "Compilation of a Countermeasure Against Instruction-Skip Fault Attacks," in Proceedings of the Third Workshop on Cryptography and Security in Computing Systems (CS2), 2016.

DYNAMIC PROTECTION: CODE POLYMORPHISM

COGITO: POLYMORPHIC RUNTIME CODE GENERATION

Definition

- **Regularly changing the behavior of a (secured) component, at runtime, while maintaining unchanged its functional properties, with runtime code generation**



COGITO: POLYMORPHIC RUNTIME CODE GENERATION

Definition

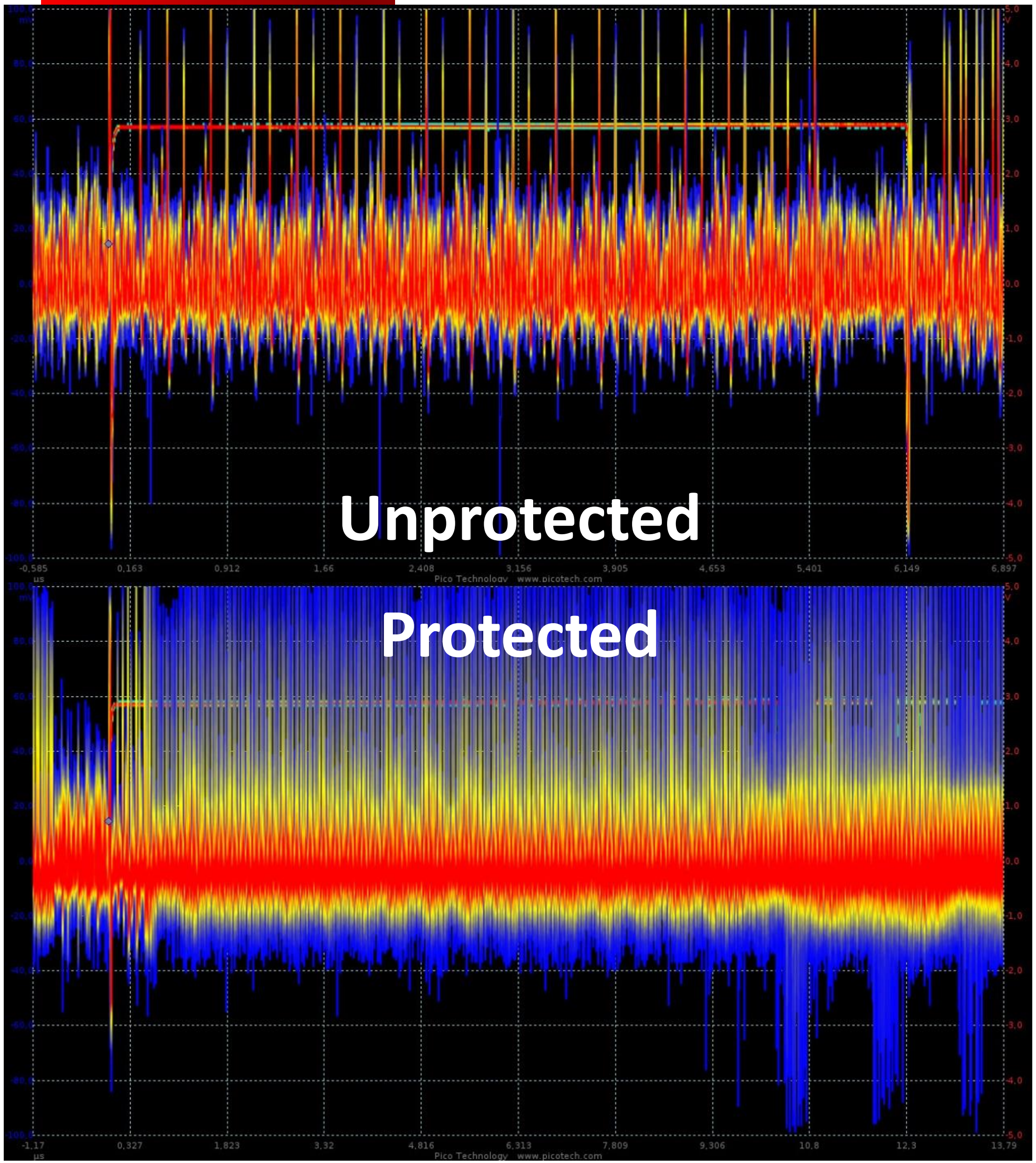
- **Regularly changing the behavior of a (secured) component, at runtime, while maintaining unchanged its functional properties, with runtime code generation**

- **Protection against physical attacks: side channel & fault attacks**
  - polymorphism changes the spatial and temporal properties of the secured code
  - Compatible with State-of-the-Art HW & SW Countermeasures

- **Protection against reverse engineering of SW**
  - the secured code is not available before runtime
  - the secured code regularly changes its form (code generation interval ω)

- **deGoal: runtime code generation for embedded systems**
  - fast code generation, tiny memory footprint

ILLUSTRATION

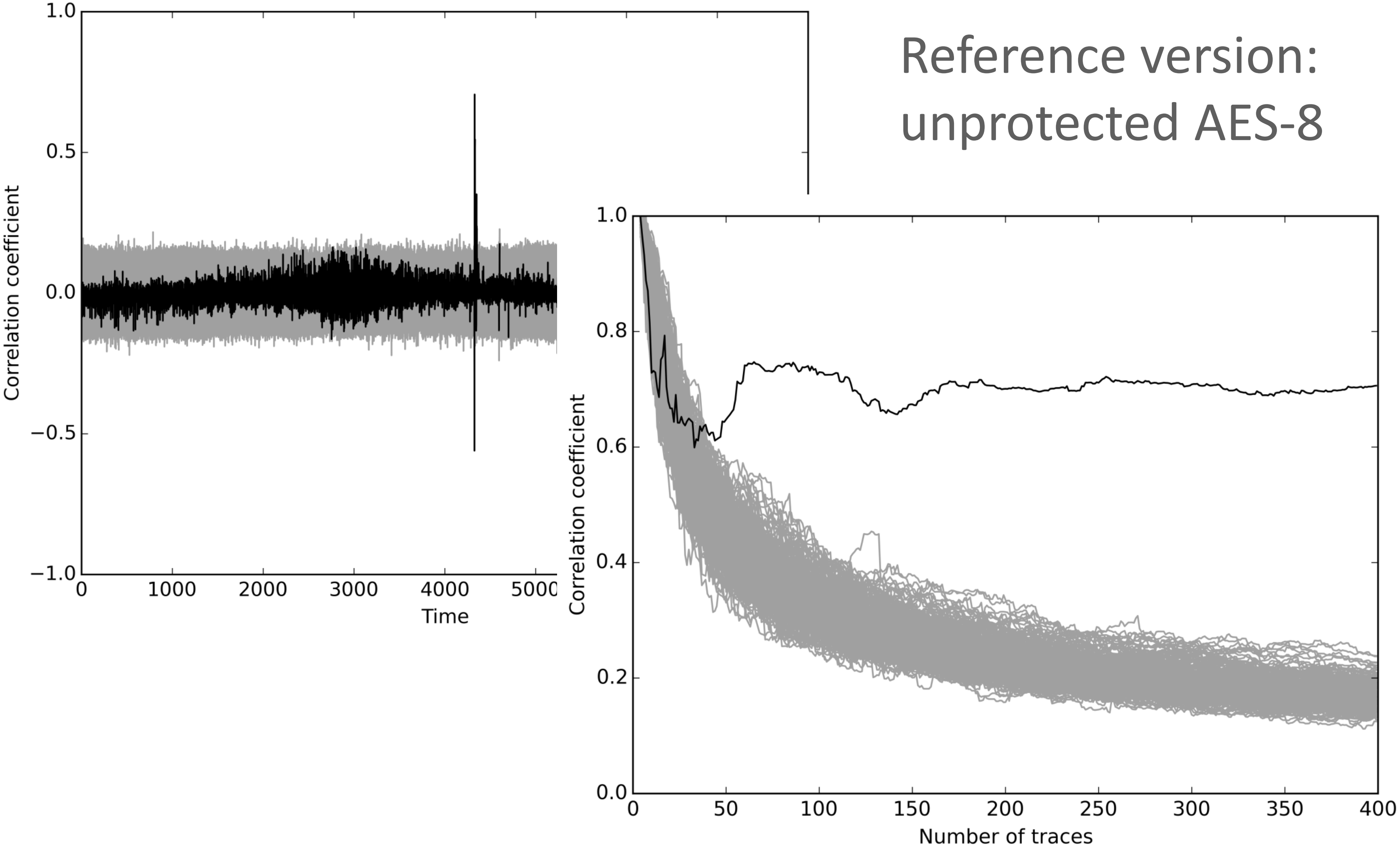


AES, 8-bit

STM32 (Cortex-M3)

IMPACT OF POLYMORPHISM ON 1ST ORDER CPA

Reference version: unprotected AES-8



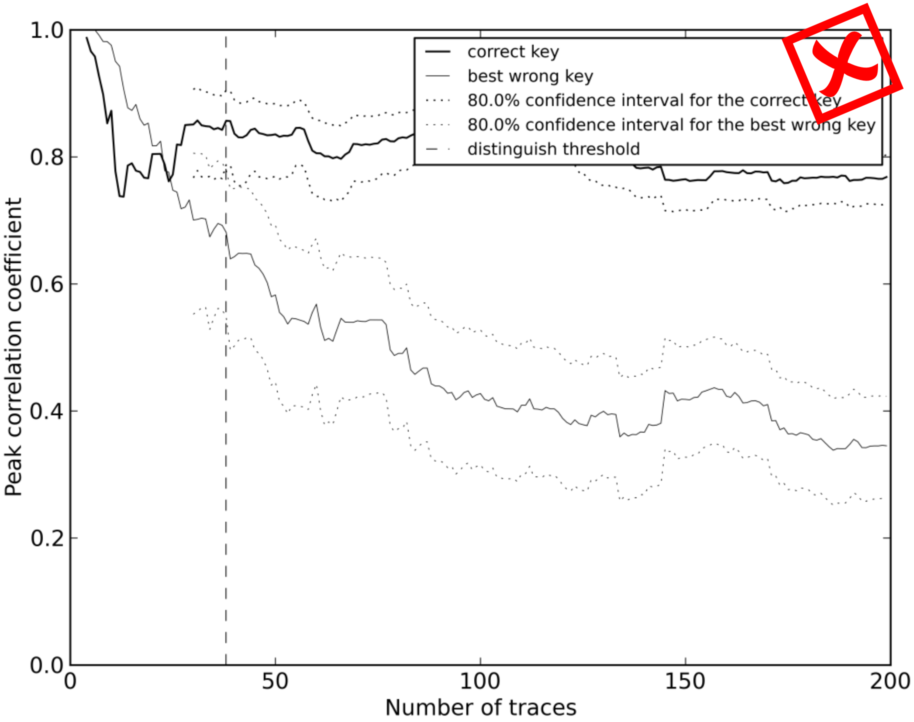
IMPACT OF POLYMORPHISM ON CPA

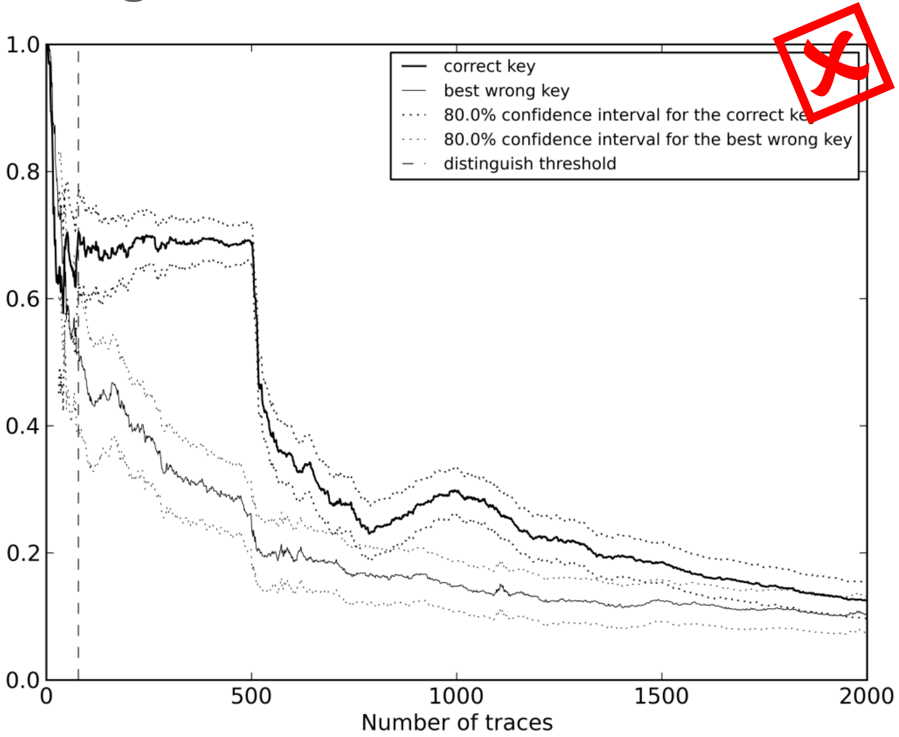
Effect of the code generation interval

Reference implementation

Polymorphic version,

code generation intervall: **500**





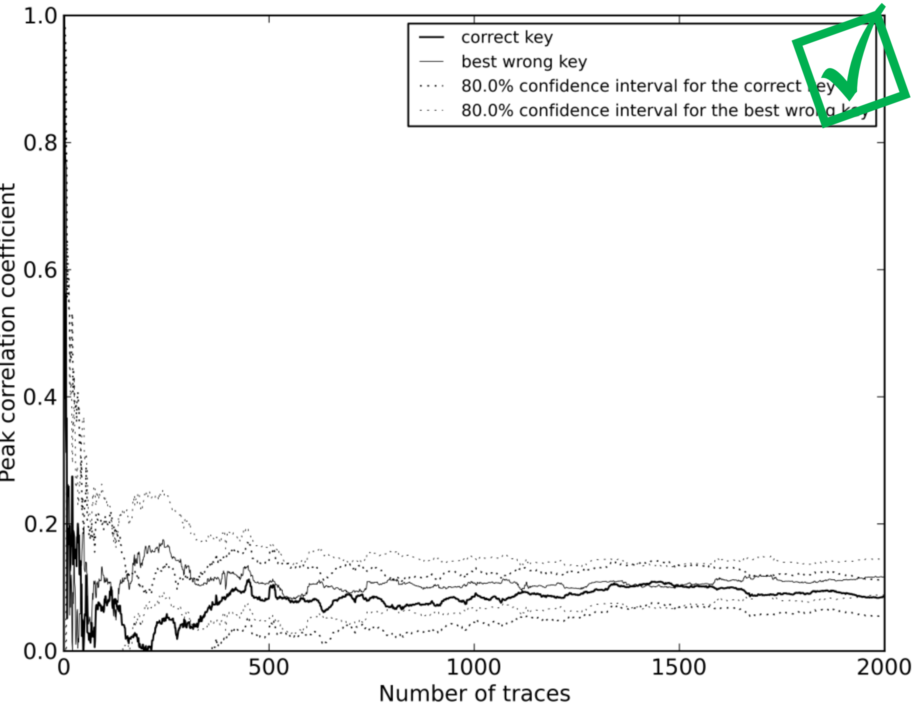
Distinguish threshold = 39 traces

Distinguish threshold = 89 traces

IMPACT OF POLYMORPHISM ON CPA

Polymorphic version

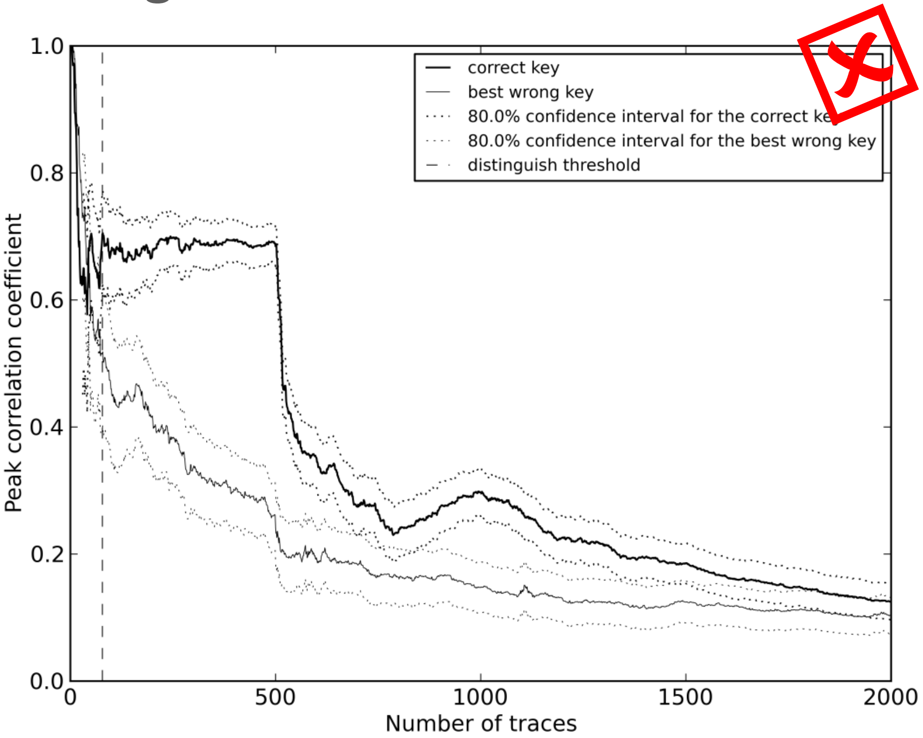
code generation interval: **20**



Distinguish threshold > 10000 traces

Polymorphic version,

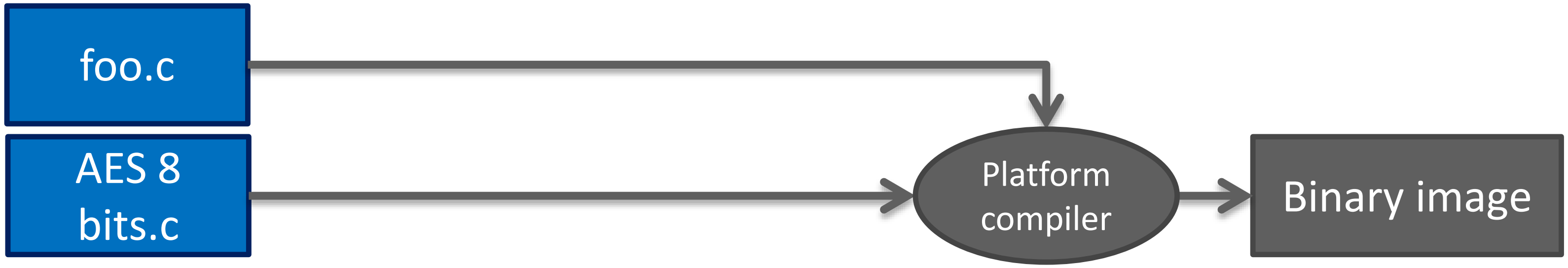
code generation intervall: **500**



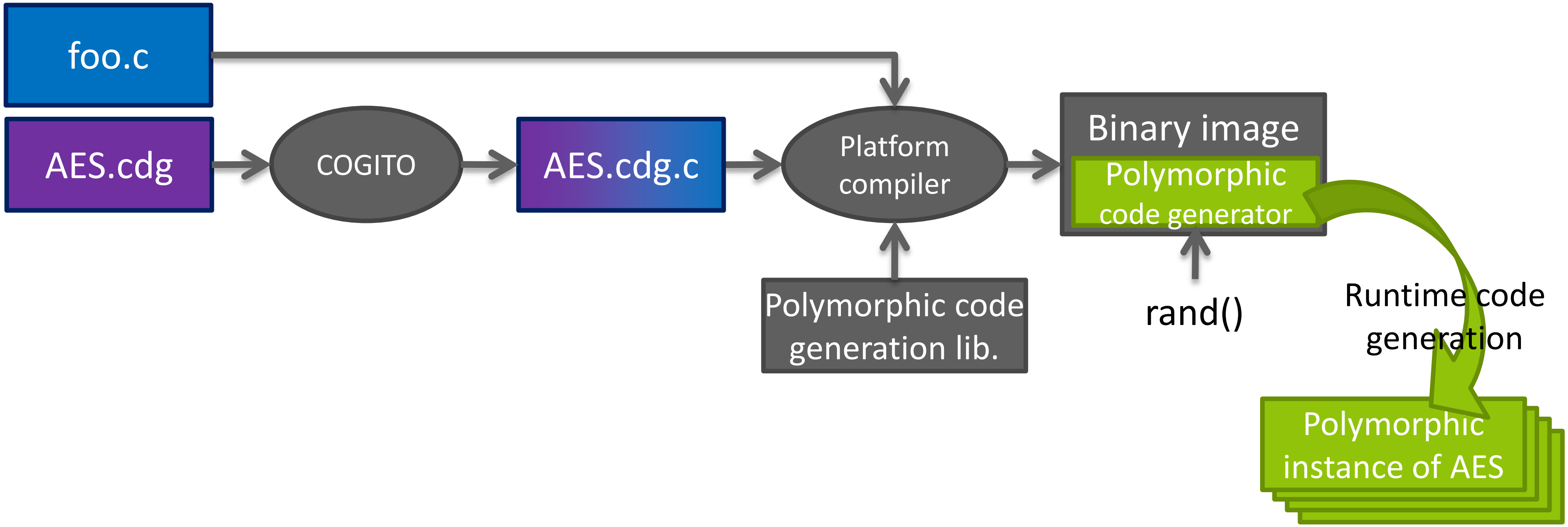
Distinguish threshold = 89 traces

IMPLEMENTATION

Reference version:



Polymorphic version, with COGITO:


foo.c
AES.cdg
COGITO
AES.cdg.c
Platform compiler
Binary image
Polymorphic code generator
Polymorphic code generation lib.
rand()
Runtime code generation
Polymorphic instance of AES

POLYMORPHIC SUBBYTES: FIRST IMPLEMENTATION

AES SubBytes: polymorphic loop

```
void subBytes_compilette(cdg_insn_t* code, const byte* sbox_addr, unsigned char* state_addr)
{
    #[
        Begin code Prelude
        Type uint32 int 32
        Alloc uint32 rstate, rstatei, rsbox, rsboxi, i
        mv rsbox, #((unsigned int)sbox_addr)
        noise_load_setup rsbox, #(256)
        mv rstate, #((unsigned int)state_addr)
    ]#
    /* insert [0; 32[ noise instructions */
    cdg_gennoise_((((PRELUDE_NOISE_LEVEL - 1) << 4) & cdg_rand()) >> 4);
    #[
        mv i, #(0)
        loop:
            lb rstatei, rstate, i      //statei = state[i]
            lb rsboxi, rsbox, rstatei  //sboxi = sbox[statei]
            sb rstate, i, rsboxi       //state[i] = sboxi
            add i, i, #(1)
            bneq loop, i, #(16)
        rtn
        End
    ]#;
}
```

Side channel leakage of key data

SUBBYTES: SAMPLES OF GENERATED MACHINE CODE

Reference version

```
stmdb     sp!, {r4, r7}
movw      r2, #33380
movt      r2, #2049
movw      r0, #44
movt      r0, #8192
movs      r4, #0
ldrb      r1, [r0, r4]  // statei = state[i]
ldrb      r3, [r2, r1]  // sboxi = sbox[statei]
strb      r3, [r0, r4]  // state[i] = sboxi
addw      r4, r4, #1
cmp       r4, #16
bne.n     0x20000852
ldmia.w   sp!, {r4, r7}
bx        lr
```

Random register allocation

Instruction substitution

Insertion of noise instructions

Instruction shuffling

No code suppression

Polymorphic instances

```
stmdb     sp!, {r5, r6, r7, r8, r9, r10, r11, lr}
movw      r12, #58220
movt      r12, #2049
movw      r2, #64
subs      r3, r3, r3
movt      r2, #8192
subs      r3, r3, r3
eor.w     r3, r3, r12
ldr.w     r3, [r12, #43]
eor.w     r3, r3, r12
movw      r9, #0
eor.w     r10, r10, r12
add.w     r10, r10, r10
sub.w     r10, r10, r10
ldr.w     r3, [r12, #207]
eor.w     r10, r10, r12
eor.w     r10, r10, r12
add.w     r10, r10, r10
ldrb.w    r0, [r2, r9]
movs      r7, #16
ldrb.w    r11, [r12, r0]
strb.w    r11, [r2, r9]
addw      r9, r9, #1
cmp       r9, r7
bne.n     0x20000894 <tmp.6949+68>
ldmia.w   sp!, {r5, r6, r7, r8, r9, r10, r11, lr}
bx        lr
```

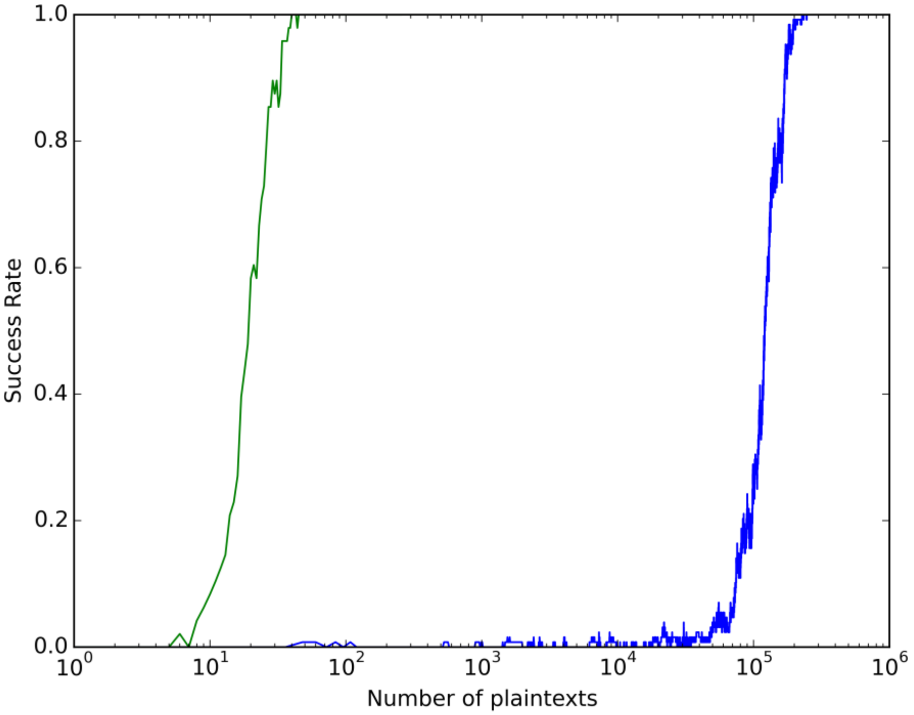
```
stmdb     sp!, {r4, r5, r6, r7, r9, r10, r11, lr}
movw      r12, #58220
movt      r12, #2049
movw      lr, #0
movw      r5, #64
movt      r5, #8192
ldr.w     r0, [r12, #128]
eor.w     r0, r0, r12
eor.w     r0, r0, r12
adds      r0, r0, r0
adds      r0, r0, r0
adds      r0, r0, r0
adds      r0, r0, r0
ldrb.w    r2, [r5, lr]
ldrb.w    r11, [r12, r2]
strb.w    r11, [r5, lr]
movs      r7, #16
addw      lr, lr, #1
cmp       lr, r7
bne.n     0x2000087c <tmp.6949+44>
ldmia.w   sp!, {r4, r5, r6, r7, r9, r10, r11, lr}
bx        lr
```

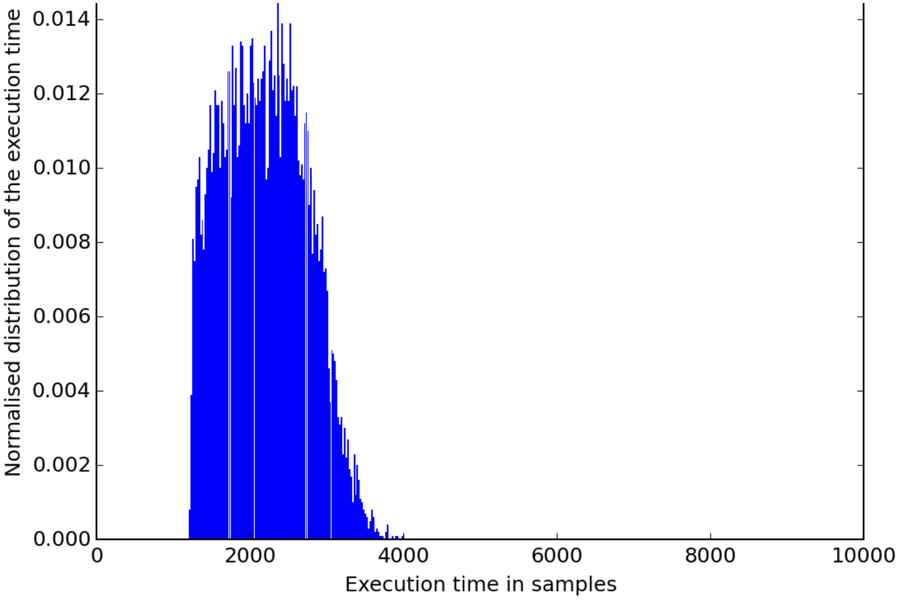
```
stmdb     sp!, {r4, r5, r6, r7,
movw      lr, #58220
movt      lr, #2049
subs      r2, r2, r2
movw      r8, #64
subs      r2, r2, r2
movt      r8, #8192
subs      r2, r2, r2
eor.w     r2, r2, lr
ldr.w     r2, [lr, #152]
ldr.w     r2, [lr, #250]
subs      r2, r2, r2
subs      r2, r2, r2
ldr.w     r2, [lr, #123]
ldr.w     r2, [lr, #158]
eor.w     r2, r2, lr
subs      r2, r2, r2
movs      r5, #0
eor.w     r2, r2, lr
ldr.w     r2, [lr, #245]
adds      r2, r2, r2
ldrb.w    r0, [r8, r5]
eor.w     r2, r2, lr
eor.w     r2, r2, lr
eor.w     r2, r2, lr
ldrb.w    r4, [lr, r0]
subs      r2, r2, r2
strb.w    r4, [r8, r5]
subs      r2, r2, r2
```

TIME DISPERSI

AES SubBytes: polymorphic loop

```
void subBytes_compilette(cdg_insn_t* code, const byte*
{
    #[
        Begin code Prelude
        Type uint32 int 32
        Alloc uint32 rstate, rstatei, rsbox, rsboxi, i
        mv rsbox, #((unsigned int)sbox_addr)
        noise_load_setup rsbox, #(256)
        mv rstate, #((unsigned int)state_addr)
    ]#
    /* insert [0; 32[ noise instructions */
    cdg_gennoise_((((PRELUDE_NOISE_LEVEL - 1) << 4) & cdg
    #[
        mv i, #(0)
        loop:
            lb rstatei, rstate, i       //statei = state[i
            lb rsboxi, rsbox, rstatei   //sboxi = sbox[sta
            sb rstate, i, rsboxi        //state[i] = sboxi
            add i, i, #(1)
            bneq loop, i, #(16)
        rtn
        End
    ]#;
}
```





TIME DISPERSION OF THE LEAKAGE POINT

AES SubBytes: polymorphic loop **+ shuffling**

```
void subBytes_compilette(cdg_insn_t* code, const byte* sbox_addr, unsigned char* state_addr)
{
  #[
    Begin code Prelude
    Type uint32 int 32
    Alloc uint32 rstate, rstatei, rsbox, rsboxi, i
    mv rsbox, #((unsigned int)sbox_addr)
    noise_load_setup rsbox, #(256)
    mv rstate, #((unsigned int)state_addr)
  ]#
  /* insert [0; 32[ noise instructions */
  cdg_gennoise_((((PRELUDE_NOISE_LEVEL - 1) << 4) & cdg_rand()) >> 4);
  int indices[SBOX_INDICES_LEN];
  init_and_permute_table(indices, SBOX_INDICES_LEN);
  for(i=0; i<16; i++) {
    #[
      Alloc uint32 rstatei, rsboxi
      lb rsboxi, rsbox, rstatei           //sboxi = sbox[statei]
      sb rstate, #(indices[i]), rsboxi    //state[i] = sboxi
      Free rstatei, rsboxi
    ]#
  }
}
```

SUBBYTES: SAMPLES OF GENERATED MACHINE CODE

Reference version

```
stmdb      sp!, {r4, r7}
movw       r2, #33380
movt       r2, #2049
movw       r0, #44
movt       r0, #8192
movs       r4, #0
ldrb       r1, [r0, r4]
ldrb       r3, [r2, r1]
strb       r3, [r0, r4]
addw       r4, r4, #1
cmp        r4, #16
bne.n      0x20000852
ldmia.w    sp!, {r4, r7}
bx         lr
```

New reference version, unrolled

```
stmdb      sp!, {r7}
movw       r1, #7723
movt       r1, #2048
movw       r0, #44
movt       r0, #8192
ldrb       r3, [r0, #0]
ldrb       r2, [r1, r3]
strb       r2, [r0, #0]
ldrb       r3, [r0, #1]
ldrb       r2, [r1, r3]
strb       r2, [r0, #1]
ldrb       r3, [r0, #2]
ldrb       r2, [r1, r3]
strb       r2, [r0, #2]
ldrb       r3, [r0, #3]
ldrb       r2, [r1, r3]
strb       r2, [r0, #3]
ldrb       r3, [r0, #4]
ldrb       r2, [r1, r3]
strb       r2, [r0, #4]
ldrb       r3, [r0, #5]
ldrb       r2, [r1, r3]
strb       r2, [r0, #5]
ldrb       r3, [r0, #6]
ldrb       r2, [r1, r3]
strb       r2, [r0, #6]
ldrb       r3, [r0, #7]
ldrb       r2, [r1, r3]
strb       r2, [r0, #7]
```

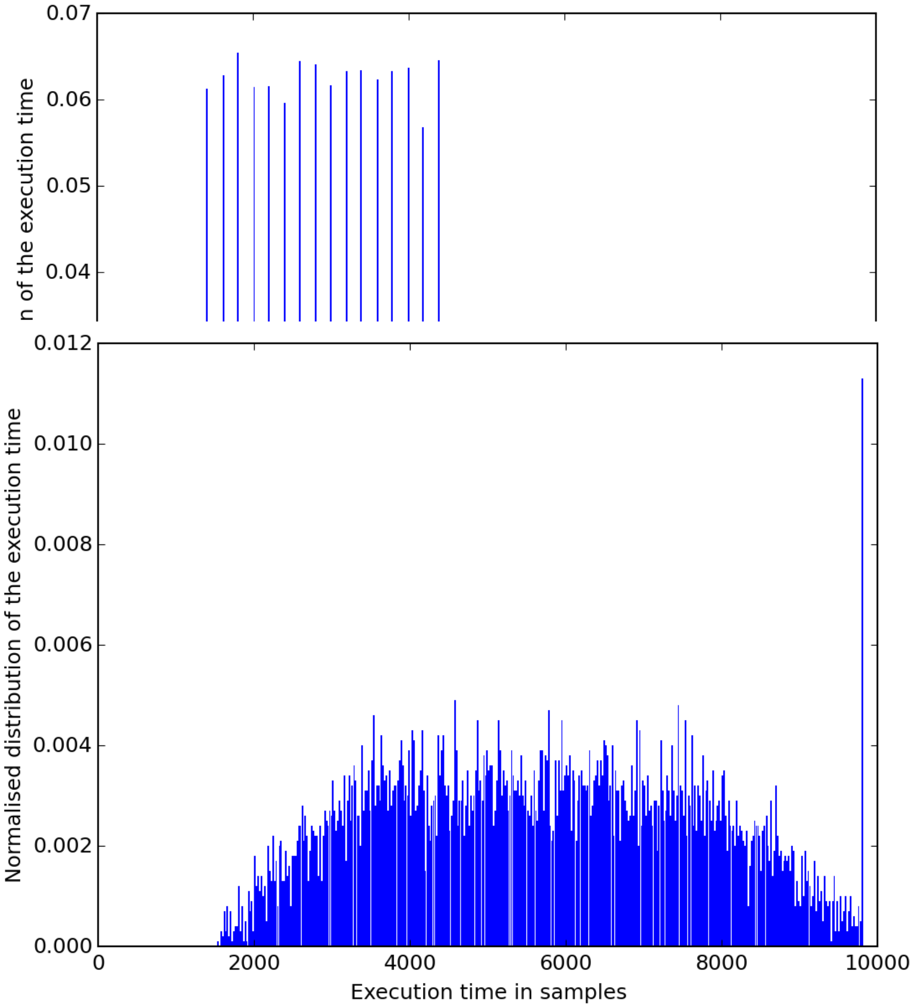
Polymorphic instance

```
stmdb      sp!, {r4, r5, r6, r7, r8, r9, r10, r11, lr}
movw       r4, #52988      ; 0xcefc
movt       r4, #2049       ; 0x801
ldr.w      r2, [r4, #193]  ; 0xc1
movw       r0, #64         ; 0x40
ldr.w      r2, [r4, #159]  ; 0x9f
movt       r0, #8192       ; 0x2000
subs       r2, r2, r2
ldr.w      r2, [r4, #125]  ; 0x7d
ldr.w      r2, [r4, #92]   ; 0x5c
eor.w      r2, r2, r4
ldr.w      r2, [r4, #118]  ; 0x76
ldr.w      r2, [r4, #192]  ; 0xc0
adds       r2, r2, r2
adds       r2, r2, r2
ldr.w      r2, [r4, #245]  ; 0xf5
eor.w      r2, r2, r4
adds       r2, r2, r2
subs       r2, r2, r2
ldrb.w     r12, [r0, #14]
subs       r2, r2, r2
adds       r2, r2, r2
ldrb.w     r5, [r4, r12]
strb       r5, [r0, #14]
ldrb.w     r9, [r0]
ldrb.w     r12, [r4, r9]
strb.w     r12, [r0]
ldrb.w     r8, [r0, #7]
```

TIME DISPERSION OF THE LEAKAGE POINT

AES SubBytes: polymorphic loop **+ shuffling**

```
void subBytes_compilette(cdg_insn_t* code, const byte* sbox_addr, unsigned char* state_addr)
{
  #[
    Begin code Prelude
    Type uint32 int 32
    Alloc uint32 rstate, rstatei, rsbox, rsboxi, i
    mv rsbox, #((unsigned int)sbox_addr)
    noise_load_setup rsbox, #(256)
    mv rstate, #((unsigned int)state_addr)
  ]#
  /* insert [0; 32[ noise instructions */
  cdg_gennoise_((((PRELUDE_NOISE_LEVEL - 1) << 4) & cdg_
  int indices[SBOX_INDICES_LEN];
  init_and_permute_table(indices, SBOX_INDICES_LEN);
  for(i=0; i<16; i++) {
    #[
      Alloc uint32 rstatei, rsboxi
      lb rsboxi, rsbox, rstatei          //sboxi = sbox
      sb rstate, #(indices[i]), rsboxi    //state[i] = s
      Free rstatei, rsboxi
    ]#
  }
}
```



Polymorphism is a *hiding* countermeasure

- The leakage instruction:
  - is not modified as compared to the same instruction in the reference version (for eval purposes)
- Unprotected AES: N < 50 traces => Polymorphic: N > 1000000 traces
  - But data leakage is still available in the traces

… compatible with masking

Compilation for cyber-security in embedded systems

Damien Couroussé
CEA – LIST / LIALP; Grenoble Université Alpes
damien.courousse@cea.fr