Compilation *and* cyber-security in embedded systems

11e rencontre de la communauté française de compilation
Aussois – 2016 – 09 – 07

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

CEA TECH DACLE

leti Grenoble

list Saclay

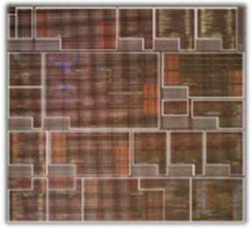
DACLE
Architectures, IC Design & Embedded Software Division

300 members
160 permanent researchers

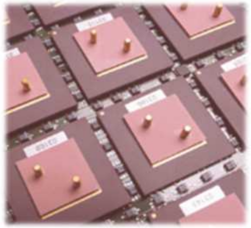
60 PhD students & postdocs

> 150 scientific papers per year

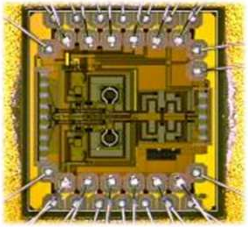
45 patents per year



Digital design



Programming



Analog & MEMs

Signal processing

Imaging

Test

LIALP

One team on code generation for performance & cybersecurity

- Runtime code generation
  - deGoal – Code specialisation with runtime code generation
  - COGITO – Code polymorphism for security in embedded components
- Compilation of countermeasures with LLVM

PRÉAMBULE

- Objectif
  - Introduction aux attaques physiques ; illustrer quelques problèmes de sécurité
  - Démystifier l'idée « *chiffrement {AES, RSA, …} = sécurité* »
  - Pourquoi il ne faut pas accorder (trop) de confiance au compilateur
    - Exemples de protections contre les fautes et le side channel
  - (Comment produire (quand même) du code sécurisé)

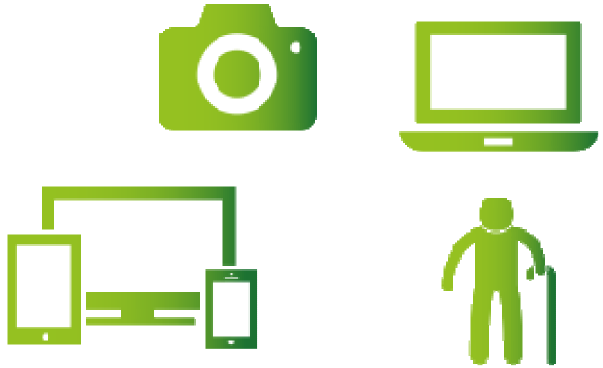
- This talk is based on *naive* examples. Their only purpose is to illustrate the ideas presented here.

BESTIARY OF EMBEDDED SYSTEMS

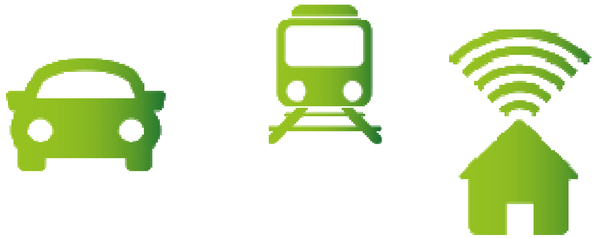
… IN NEED FOR SECURITY CAPABILITES

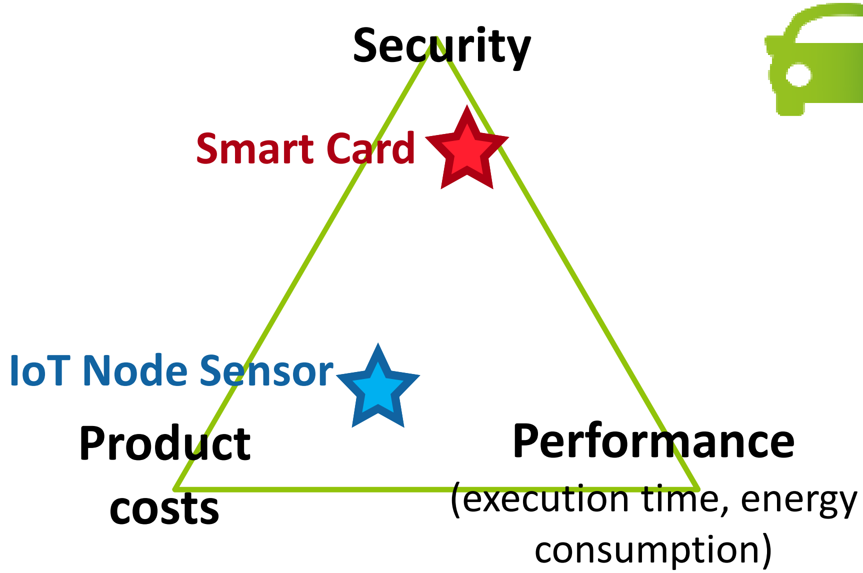
Smart Card

Secure Element inside…

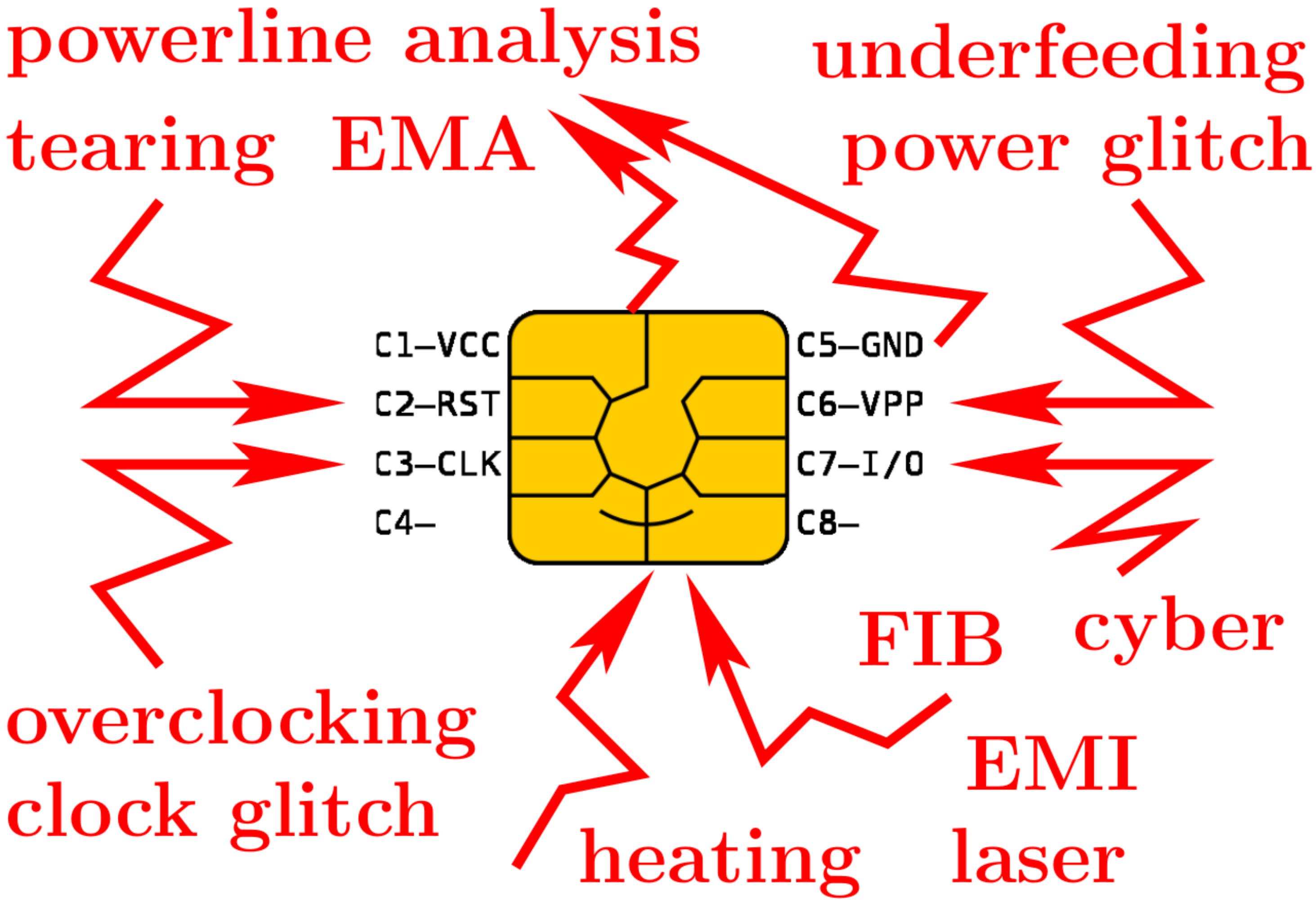


… And many other things





CYBER-PHYSICAL ATTACKS



Courtesy of Sylvain Guilley, Télécom ParisTech - Secure-IC

INTRODUCTION AUX ATTAQUES PHYSIQUES

L'attaquant procède en deux temps :

1. analyse globale, caractérisation de la cible et recherche de faiblesses
2. attaque ciblée sur un point de faiblesse

… ou pas

Procédés mis en œuvre :

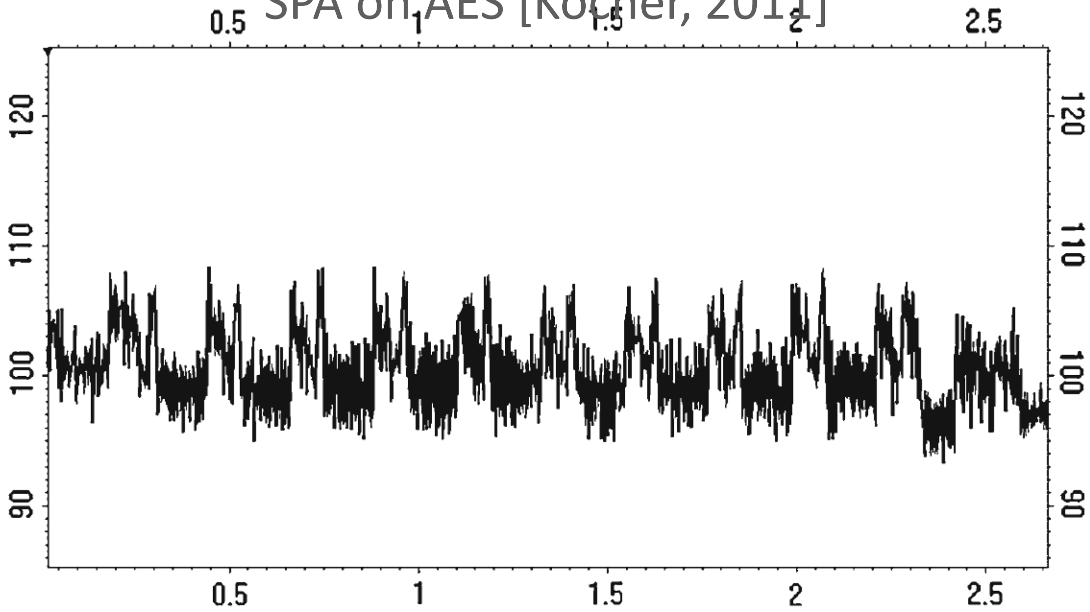
- **Cryptanalyse**
  En dehors du périmètre de la présentation
- **Rétro-conception / reverse engineering**
  Inspection matérielle : décapsulation, abrasion, etc.
  Inspection logicielle : debug, memory dumps, analyse de code, etc.
  Dynamique (SPA, SCARE, FIRE…), à l'aide des attaques ci-dessous
- **Attaques passives : attaques par canaux cachés**
  Observations électromagnétiques, électriques, acoustiques, temps d'exécution, etc.
- **Attaques actives : attaques en fautes**
  Sur/sous-alimentation, laser, illumination, corruption horloge, etc.

- Bonus : **Attaques logiques**
  En dehors du périmètre de cette présentation
  Un problème réglé pour la Haute Sécurité ?

SIDE CHANNEL / SIMPLE POWER ANALYSIS (SPA)

Extraction of information by manual inspection of side-channel traces

SPA on AES [Kocher, 2011]



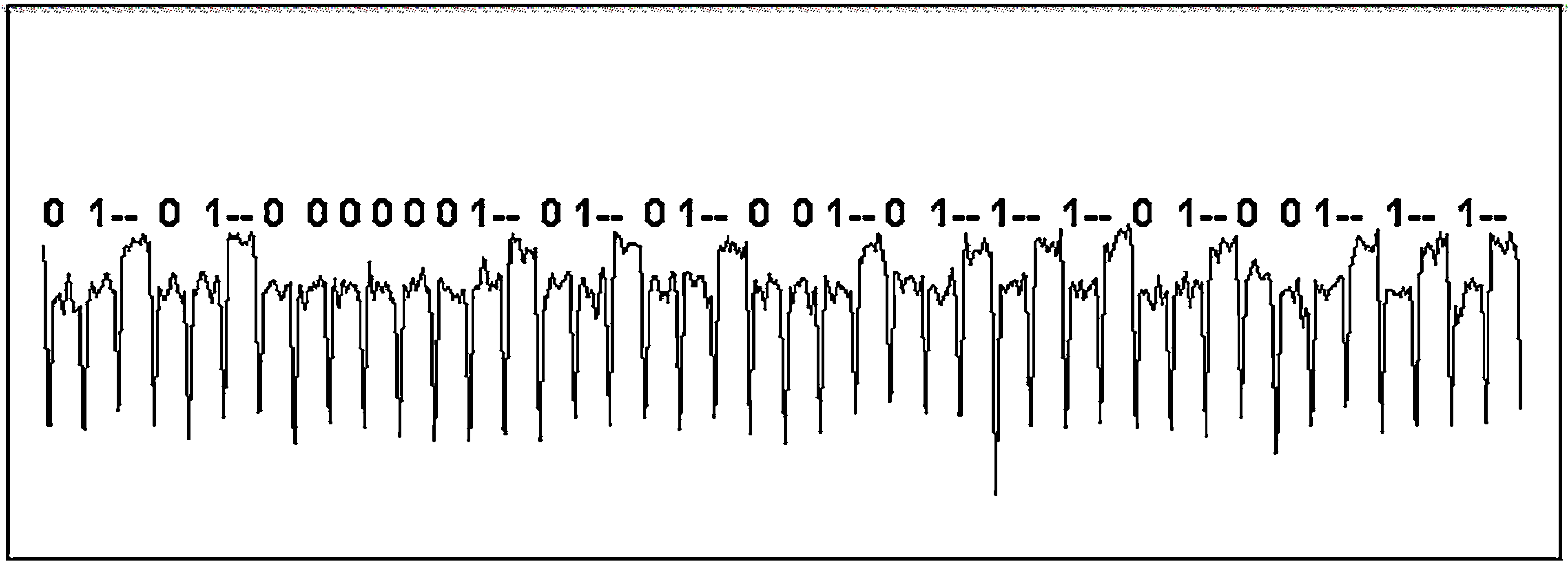
The AES rounds are « clearly » visible

- 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.

A Lightbulb Worm? A teardown of the Philips Hue.
Colin O'Flynn. BlackHat 2016. cf. slides ~60 to 70

SIDE CHANNEL / SIMPLE POWER ANALYSIS (SPA)

SPA on RSA [Kocher, 2011]



Direct access to key contents:

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

SIDE CHANNEL / DIFFERENTIAL/CORRELATION POWER ANALYSIS (DPA/CPA)

Finding a needle in a haystack…

- Relationship between the different components of power consumption:

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

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

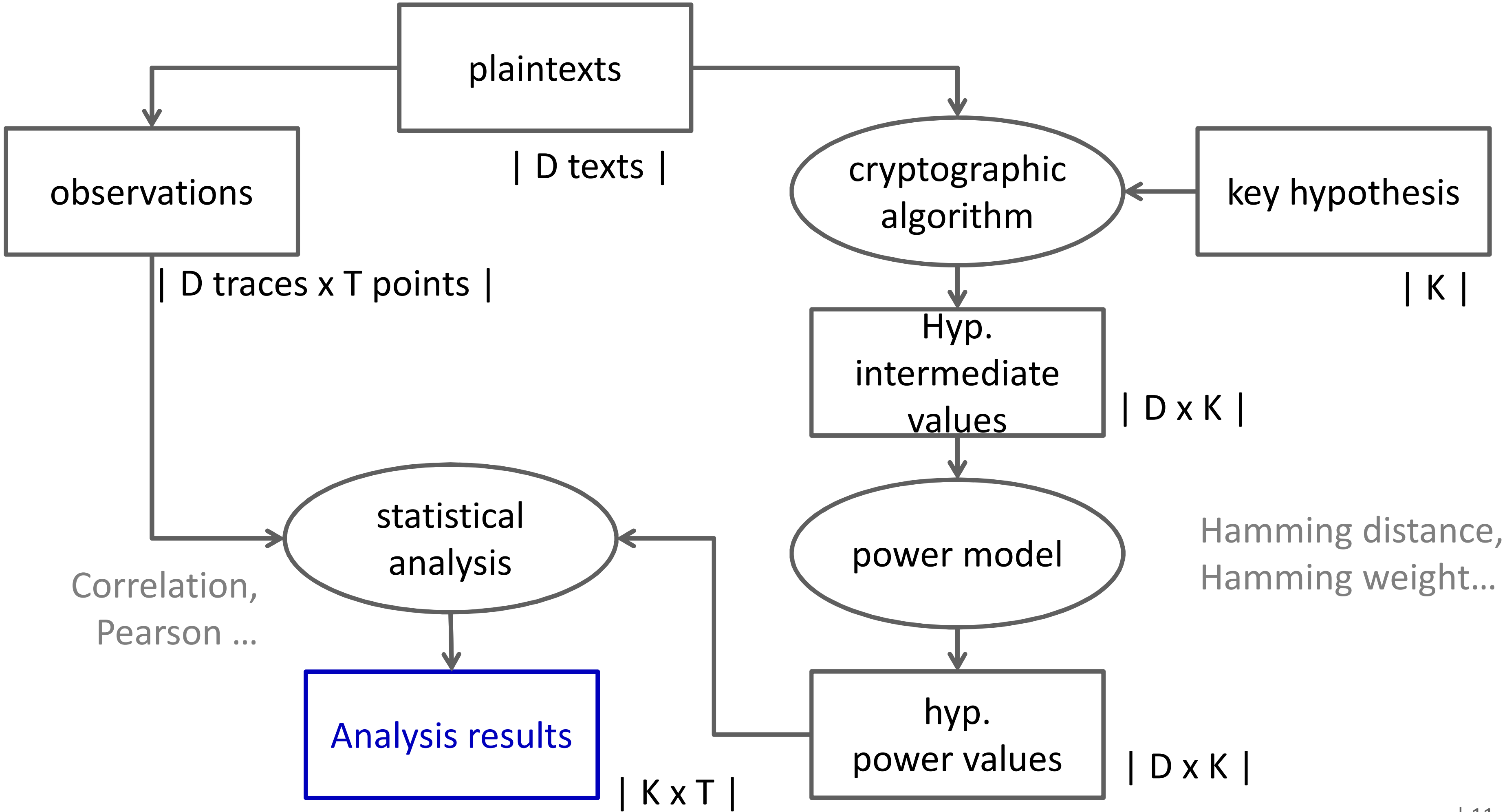
needle: $P_{exploitable}$

haystack: $P_{switching.noise} + P_{electronic.noise} + P_{const}$

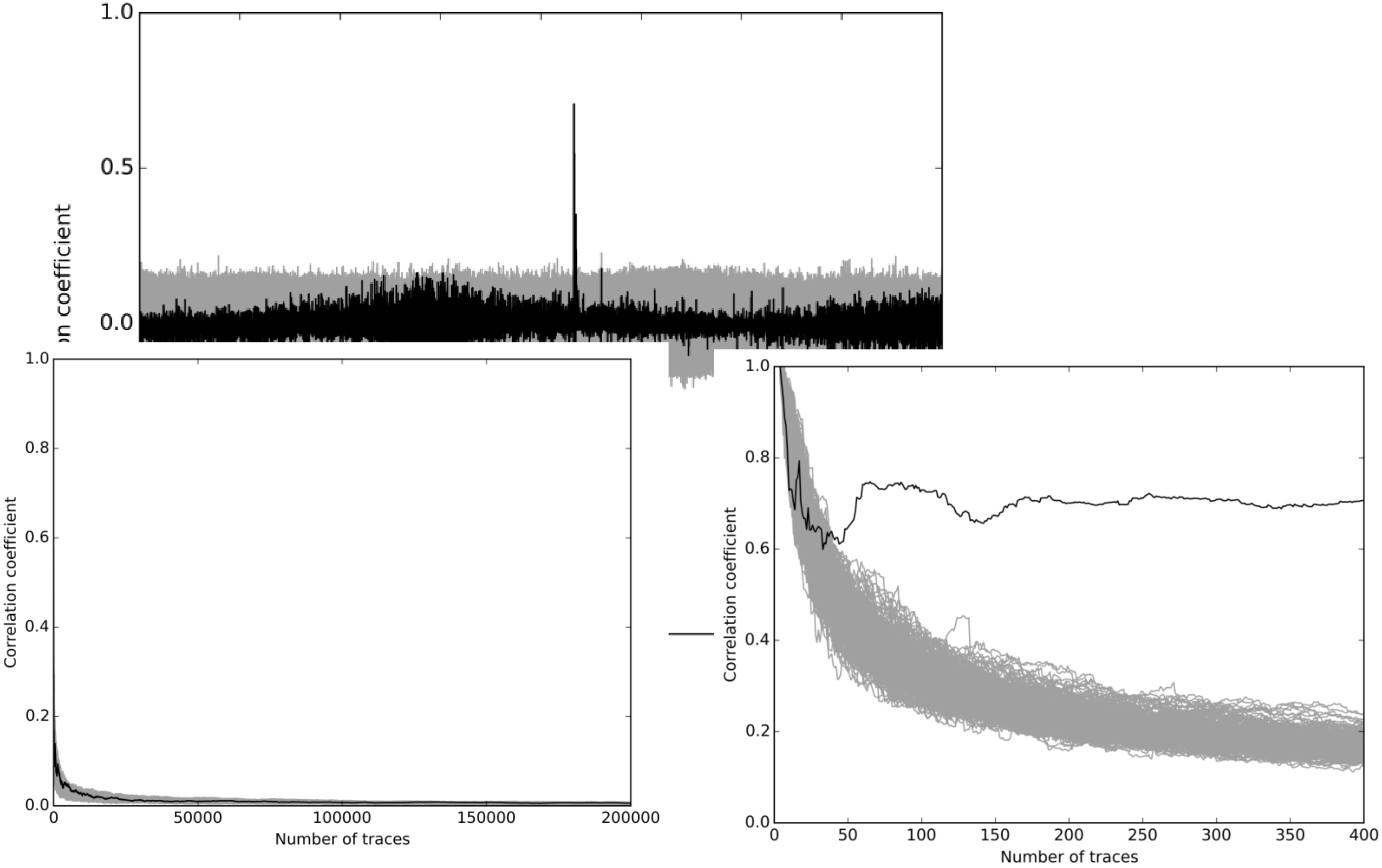
- Other needles & stacks
  - Electromagnetic emissions
  - Execution time
  - Chip temperature
  - Etc.

CPA IN PRACTICE

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



CPA IN PRACTICE



FAULTS

Attack models

- Logic
  - Memory bit flip (set, reset, toggle)
- Instruction level
  - Instruction skip
  - Instruction alteration -> jumps
- …

Attacks

- Control-flow hijacking
- Differential Faults Analysis

  normal execution: f(k, m) –> c

  faulted execution: f'(k, m) -> c'

  differential analysis: Δ(c, c') ~> k
- Ineffective Fault Analysis / safe error

  « the fault injection capability may be used as a probing tool »
- Reverse engineering (FIRE. Fault Injection for Reverse Engineering)

COMPILATION CLASSIQUE

ET

IMPACT SUR LA SÉCURITÉ

(FAUT-IL FAIRE CONFIANCE AU COMPILATEUR ?)

A MUST READ

TURING AWARD LECTURE

Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

Communications of the ACM
August 1984,
vol 28 number 8

KEN THOMPSON

INTRODUCTION

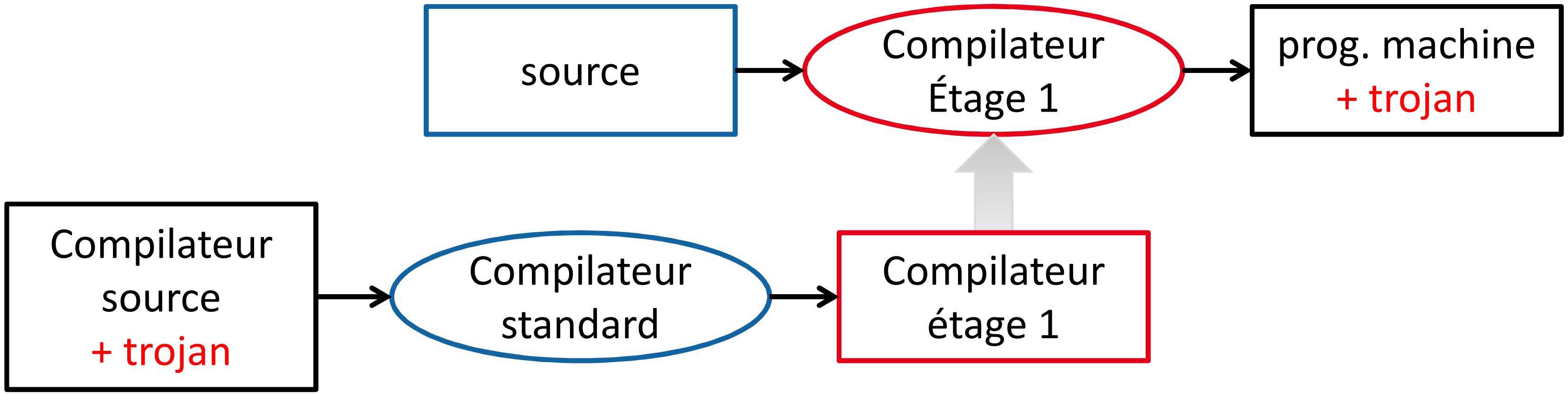
I thank the ACM for this award. I can't help but feel that I am receiving this honor for timing and serendipity as much as technical merit. UNIX[1] swept into popularity with an industry-wide change from central mainframes to autonomous minis. I suspect that Daniel Bobrow [1] would be here instead of me if he could not afford a PDP-10 and had had to "settle" for a PDP-11. Moreover, the current state of UNIX is the result of the labors of a large number of people.

There is an old adage, "Dance with the one that brought you," which means that I should talk about programs. I would like to present to you the cutest program I ever wrote. I will do this in three stages and try to bring it together at the end.

STAGE I

In college, before video games, we would amuse ourselves by posing programming exercises. One of the favorites was to write the shortest self-reproducing program. Since this is an exercise divorced from reality, the usual vehicle was FORTRAN. Actually, FORTRAN was the language of choice for the same reason that three-legged races are popular.

REFLECTIONS ON TRUSTING TRUST.
INSERTION SILENCIEUSE DE TROJANS



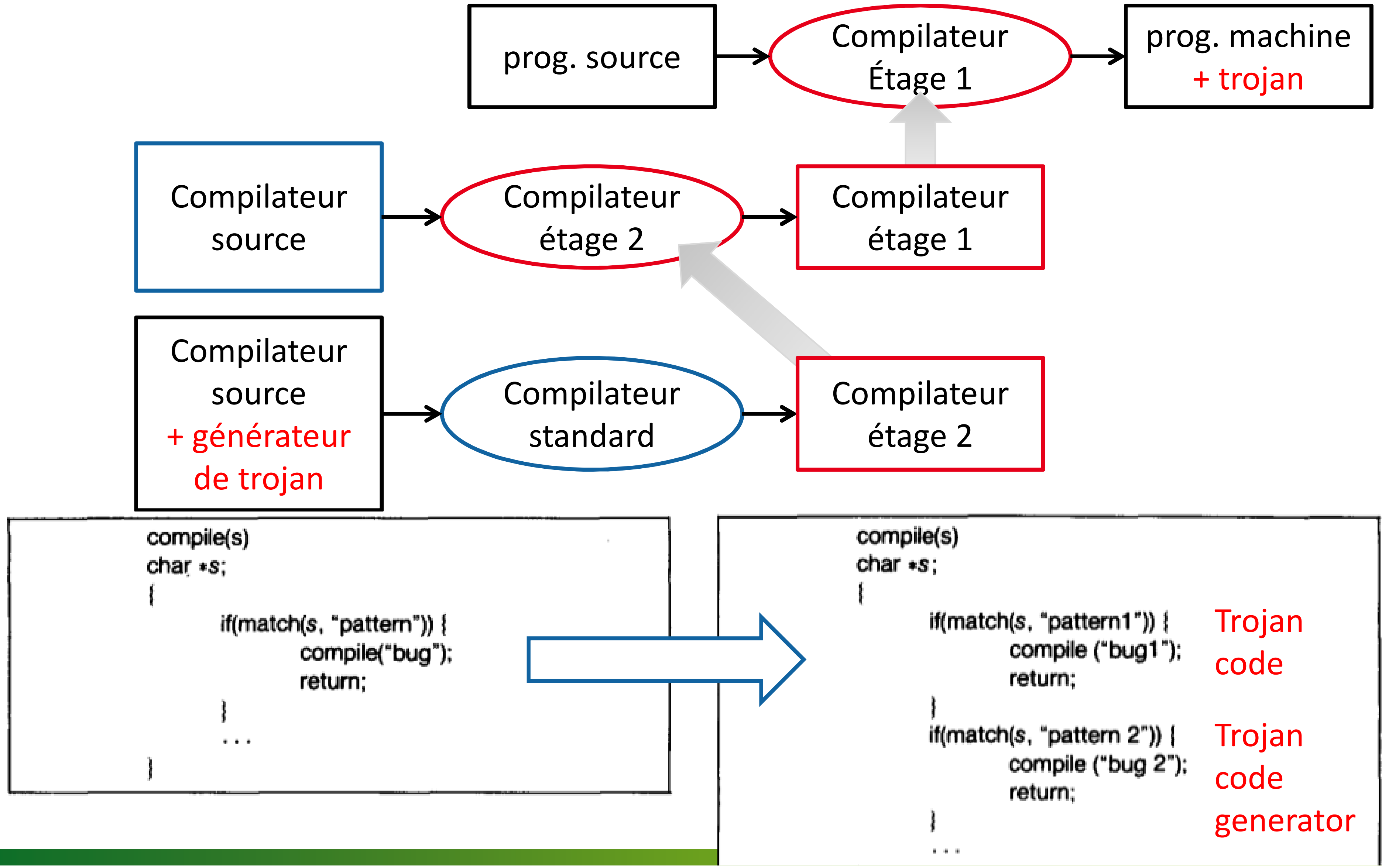
Compilateur étage 1:

```
compile(s)
char *s;
{
        ...
}
```

```
compile(s)
char *s;
{
        if(match(s, "pattern")) {
                compile("bug");
                return;
        }
        ...
}
```

Trojan
code

REFLECTIONS ON TRUSTING TRUST.
INSERTION SILENCIEUSE DE TROJANS



DEVOIRS D'UN COMPILATEUR

- **Devoirs**
  - Garantir l'équivalence fonctionnelle prog. source -> prog. machine
    - « fonctionnel »/« fonctionnalité » est un terme vague et difficile à décrire
      - Effets de bords ?
      - Déterminisme temporel pour le temps réel ?
      - Évaluation paresseuse ?
    - Pas de garantie formelle,
      - Quelques exceptions, par exemple CompCert.
    - Pas d'exactitude (*correctness*) par construction (en tout cas pas en C)
    - Encore faut-il que le programme écrit par le développeur soit correct…
- **Objectif: optimiser un ou plusieurs critères de performance**
  - **Temps d'exécution**
  - Ressources: taille du programme
  - Énergie, consommation électrique, puissance
  - Il n'existe pas de critère complet pour l'optimalité ou la convergence
    - Nature de l'algorithme
    - Architecture / micro-architecture
    - Données

- **Droits**
  - Réorganiser le programme cible, en respectant la sémantique du programme décrite par le développeur
    - opérations machines, blocs de base
  - Choix de la meilleure traduction code source --> code machine
  - Ne pas conserver tout le code écrit par le développeur (s’il ne participe pas au calcul du résultat final)

- **Quelques Passes d’optimisation:**
  - *dead code elimination*
  - *global value numbering*
  - common-subexpression elimination
  - *strength reduction*
  - *loop strength reduction, loop simplification, loop-invariant code motion*
  - Etc.

- **LLVM’s Analysis and Transform Passes , le 30/06/2016**
  - 40 passes d’analyse
  - 56 passes de transformation
  - 10 passes utilitaires
  - … backends, etc.

COMPILATION CLASSIQUE
ET
IMPACT SUR LA SÉCURITÉ

ATTAQUES EN FAUTES

```c
typedef uint32_t bool_t;
typedef uint8_t  byte_t;

#define true  0xAA
#define false 0x66

#define SIZE_OF_PIN 4

byte_t pin[SIZE_OF_PIN]; // is initialized elsewhere
byte_t user[SIZE_OF_PIN];

bool_t verify(byte_t buffer[SIZE_OF_PIN])
{
  size_t i;
  bool_t diff = false;

  bool_t status = false;

  for (i=0; i<SIZE_OF_PIN; i++) {
    if (buffer[i] != pin[i]) {
      diff = true;
    }
  }

  if ((SIZE_OF_PIN == i) && (false == diff)) {
    status = true;
  }

  return status;
}

int main(void)
{
   size_t i;
  for (i=0; i<SIZE_OF_PIN; i++) {
    pin[i] = i;
  }
```

Compilation en –O0:

```
Dump of assembler code for function verify:
   0x000084e4 <+0>:          push {r11}              ; (str r11, [sp, #-4
   0x000084e8 <+4>:          add r11, sp, #0
   0x000084ec <+8>:          sub sp, sp, #28
   0x000084f0 <+12>:         str r0, [r11, #-24]
   0x000084f4 <+16>:         mov r3, #102            ; 0x66
   0x000084f8 <+20>:         str r3, [r11, #-12]
   0x000084fc <+24>:         mov r3, #102            ; 0x66
   0x00008500 <+28>:         str r3, [r11, #-16]
   0x00008504 <+32>:         mov r3, #0
   0x00008508 <+36>:         str r3, [r11, #-8]
   0x0000850c <+40>:         b 0x854c <verify+104>
   0x00008510 <+44>:         ldr r2, [r11, #-24]
   0x00008514 <+48>:         ldr r3, [r11, #-8]
```

Extrait du manuel gcc :

-O0: This level (that is the letter "O" followed by a zero) turns off optimization entirely and is the default if no -O level is specified in CFLAGS or CXXFLAGS. This reduces compilation time and can improve debugging info, but some applications will not work properly without optimization enabled. This option is not recommended except for debugging purposes.

```
-8]
ify+44>
-8]
   0x0000855c <+120>:        cmp r3, #4
```

Lionel Rivière, 2015. Securing software implemenations against fault injection attacks on embedded systems. Thèse de doctorat. Telecom ParisTech.
M. Wittemann, 2012. Side channel security for embedde software. ESC 2012.

```c
typedef uint32_t bool_t;
typedef uint8_t  byte_t;

#define true  0xAA
#define false 0x66

#define SIZE_OF_PIN 4

byte_t pin[SIZE_OF_PIN]; // is initialized elsewhere
byte_t user[SIZE_OF_PIN];

bool_t verify(byte_t buffer[SIZE_OF_PIN])
{
  size_t i;
  bool_t diff = false;

  bool_t status = false;

  for (i=0; i<SIZE_OF_PIN; i++) {
    if (buffer[i] != pin[i]) {          #1
      diff = true;
    }
  }

  if ((SIZE_OF_PIN == i) && (false == diff)) {   #2
    status = true;
  }

  return status;
}

int main(void)
{
   size_t i;
  for (i=0; i<SIZE_OF_PIN; i++) {
    pin[i] = i;
  }
```

Compilation en –Os:

```
Dump of assembler code for function verify:
   0x00008518 <+0>:  push     {r4, lr}
   0x0000851c <+4>:  ldr      r4, [pc, #48]  ; 0x8554 <verify+60>
   0x00008520 <+8>:  mov      r2, #102       ; diff <- false
   0x00008524 <+12>: mov      r3, #0         ; i <- 0
   0x00008528 <+16>: ldrb     r12, [r0, r3]  ; r12 <- buffer[i]
   0x0000852c <+20>: ldrb     r1, [r3, r4]   ; r1 <- pin[i]
   0x00008530 <+24>: add      r3, r3, #1     ; i <- i+1
   0x00008534 <+28>: cmp      r12, r1        ; r12 ?= r1
   0x00008538 <+32>: movne    r2, #170       ; diff <- true          #1(x4)
   0x0000853c <+36>: cmp      r3, #4         ; i ?= SIZE_OF_PIN
   0x00008540 <+40>: bne      0x8528 <verify+16>
   0x00008544 <+44>: cmp      r2, #102       ; diff ?= false         #2
   0x00008548 <+48>: moveq    r0, #170       ; status <- true
   0x0000854c <+52>: movne    r0, #102       ; status <- false
   0x00008550 <+56>: pop      {r4, pc}
   0x00008554 <+60>: andeq    r0, r1, r9, ror #14
End of assembler dump.
```

Il manque déjà un test!!!
Lequel ?

VERIFY PIN

Compilation en –O3:

```c
typedef uint32_t bool_t;
typedef uint8_t  byte_t;

#define true  0xAA
#define false 0x66

#define SIZE_OF_PIN 4

byte_t pin[SIZE_OF_PIN]; // is initialized elsewhere
byte_t user[SIZE_OF_PIN];

bool_t verify(byte_t buffer[SIZE_OF_PIN])
{
  size_t i;
  bool_t diff = false;

  bool_t status = false;

  for (i=0; i<SIZE_OF_PIN; i++) {
    if (buffer[i] != pin[i]) {
      diff = true;
    }
  }

  if ((SIZE_OF_PIN == i) && (false == diff)) {
    status = true;
  }

  return diff;
}

int main(void)
{
   size_t i;
  for (i=0; i<SIZE_OF_PIN; i++) {
    pin[i] = i;
  }

  return verify(user);
}
```

```
Dump of assembler code for function verify:
   0x00008504 <+0>:  ldr r3, [pc, #100]      ;        r3 <- pin[]
   0x00008508 <+4>:  push {r4, r5}
   0x0000850c <+8>:  ldrb r2, [r0]           ;        r2  <- user[0]
   0x00008510 <+12>: ldrb r12, [r3]          ;        r12 <- pin[0]
   0x00008514 <+16>: ldrb r1, [r0, #1]       ;        r1  <- user[1]
   0x00008518 <+20>: ldrb r5, [r3, #1]       ;        r5  <- user[1]
   0x0000851c <+24>: cmp r12, r2             ; user[0] ?= pin[0]
   0x00008520 <+28>  move r2, #102           ; OK  => r2  <- 0x66
   0x00008524 <+32>  ldrb r4, [r0, #2]       ;        r4  <- user[2]
   0x00008528 <+36>  ldrb r12, [r3, #2]      ;        r12 <- pin[2]
   0x0000852c <+40>  movne r2, #170          ; NOK => r2  <- 0xAA
   0x00008530 <+44>: cmp r1, r5              ; user[1] ?= pin[1]
   0x00008534 <+48>: ldrb r0, [r0, #3]       ;        r0  <- user[3]
   0x00008538 <+52>  moveq r1, r2            ; OK  => r1  <- r2  // ???
   0x0000853c <+56>: ldrb r2, [r3, #3]       ;        r2  <- pin[3]
   0x00008540 <+60>: movne    r1, #170       ; NOK => r1  <- 0xAA
   0x00008544 <+64>: cmp      r4, r12        ; user[2] ?= pin[2]
   0x00008548 <+68>: moveq    r3, r1         ; OK  => r3  <- r1 // ???
   0x0000854c <+72>: movne    r3, #170       ; NOK => r3  <- 0XAA
   0x00008550 <+76>: cmp      r0, r2
   0x00008554 <+80>: moveq    r0, r3
   0x00008558 <+84>: movne    r0, #170       ; 0xaa
   0x0000855c <+88>  cmp      r0, #102       ; 0x66
   0x00008560 <+92>  moveq    r0, #170       ; 0xaa
   0x00008564 <+96>  movne    r0, #102       ; 0x66
   0x00008568 <+100>:pop      {r4, r5}
   0x0000856c <+104>:bx       lr
   0x00008570 <+108>:andeq    r0, r1, r8, lsl #15
End of assembler dump.
```

VERIFY PIN -> DOUBLE IF ?

```c
typedef uint32_t bool_t;
typedef uint8_t  byte_t;

#define true  0xAA
#define false 0x66

#define SIZE_OF_PIN 4

byte_t pin[SIZE_OF_PIN]; // is initialized elsewhere
byte_t user[SIZE_OF_PIN];

bool_t verify(byte_t buffer[SIZE_OF_PIN])
{
  size_t i;
  bool_t diff = false;

  bool_t status = false;

  for (i=0; i<SIZE_OF_PIN; i++) {
    if (buffer[i] != pin[i]) {
      diff = true;
    }
  }
  if ((SIZE_OF_PIN == i) && (false == diff)) {
      status = true;
  }

  for (i=0; i<SIZE_OF_PIN; i++) {
    if (buffer[i] != pin[i]) {
      diff = true;
    }
  }
  if ((SIZE_OF_PIN == i) && (false == diff)) {
      status = true;
  }

  return status;
}

int main(void)
```

Compilation en –Os:

```
Dump of assembler code for function verify:
   0x00008518 <+0>:    push    {r4, lr}
   0x0000851c <+4>:    ldr     r4, [pc, #48]   ; 0x8554 <verify+60>
   0x00008520 <+8>:    mov     r2, #102        ; 0x66
   0x00008524 <+12>:   mov     r3, #0
   0x00008528 <+16>:   ldrb    r12, [r0, r3]   ; r12 <- buffer[i]
   0x0000852c <+20>:   ldrb    r1, [r3, r4]    ; r1 <- pin[i]
   0x00008530 <+24>:   add     r3, r3, #1      ; i <- i+1
   0x00008534 <+28>:   cmp     r12, r1         ; r12 ?= r1
   0x00008538 <+32>:   movne   r2, #170        ; 0xaa
   0x0000853c <+36>:   cmp     r3, #4
   0x00008540 <+40>:   bne     0x8528 <verify+16>
   0x00008544 <+44>:   cmp     r2, #102        ; 0x66
   0x00008548 <+48>:   moveq   r0, #170        ; 0xaa
   0x0000854c <+52>:   movne   r0, #102        ; 0x66
   0x00008550 <+56>:   pop     {r4, pc}
   0x00008554 <+60>:   andeq   r0, r1, r9, ror #14
End of assembler dump.
```

ser –O0 ?

émentation assembleur ?

ser des « recettes » de cuisine ?

COMPILATION CLASSIQUE
ET
IMPACT SUR LA SÉCURITÉ

CANAUX CACHÉS

INSERTION OF DUMMY INSTRUCTIONS

- **Insertion statique d'une routine de désynchronisation:**

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

    for(i = 0; i<16; i+=4)
    {
        CORON();
        state[i+0] = sbox[ state[i+0] ];
        state[i+1] = sbox[ state[i+1] ];
        state[i+2] = sbox[ state[i+2] ];
        state[i+3] = sbox[ state[i+3] ];
    }
}
```

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

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

    nbIt_Coron++;
}
```

- **Même effet possible aussi avec un timer et un gestionnaire d'interruptions**

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

Compilation en –Os:

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

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

    nbIt_Coron++;
}
```

```
Dump of assembler code for function noiseCoron:
   0x0000859c <+0>:  push     {r4, lr}
   0x000085a0 <+4>:  ldr      r4, [pc, #28] ; <noiseCoron+40>
   0x000085a4 <+8>:  ldr      r3, [r4]
   0x000085a8 <+12>: cmp      r3, #160  ; 0xa0
   0x000085ac <+16>: bne      0x85b4 <noiseCoron+24>
   0x000085b0 <+20>: bl       0x8524 <genNoiseCoron>
   0x000085b4 <+24>: ldr      r3, [r4]
   0x000085b8 <+28>: add      r3, r3, #1
   0x000085bc <+32>: str      r3, [r4]
   0x000085c0 <+36>: pop      {r4, pc}
   0x000085c4 <+40>: andeq    r0, r1, r0, lsr r8
End of assembler dump.
```

FROM RESEARCH TO INDUSTRY
cea tech

INSERTION OF DUMMY INSTRUCTIONS

Compilation en –Os:

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

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

    nbIt_Coron++;
}
```

```
ıp of assembler code for function noiseCoron:
 0x0000859c <+0>:  push  {r4, lr}
 0x000085a0 <+4>:  ldr   r4, [pc, #60]      ; <noiseCoron+72>
 0x000085a4 <+8>:  ldr   r3, [r4]
 0x000085a8 <+12>: cmp   r3, #160    ; 0xa0
 0x000085ac <+16>: bne   0x85b4 <noiseCoron+24>
 0x000085b0 <+20>: bl    0x8524 <genNoiseCoron>
 0x000085b4 <+24>: ldr   r3, [pc, #44]      ; <noiseCoron+76>
 0x000085b8 <+28>: ldr   r2, [r4]
 0x000085bc <+32>: ldr   r1, [r3, r2, lsl #2]
 0x000085c0 <+36>: mov   r3, #0
 0x000085c4 <+40>: cmp   r3, r1
 0x000085c8 <+44>: beq   0x85d8 <noiseCoron+60>
 0x000085cc <+48>: add   r3, r3, #1
 0x000085d0 <+52>: nop
 0x000085d4 <+56>: b     0x85c4 <noiseCoron+40>
 0x000085d8 <+60>: add   r2, r2, #1
 0x000085dc <+64>: str   r2, [r4]
 0x000085e0 <+68>: pop   {r4, pc}
 0x000085e4 <+72>: andeq r0, r1, r4, asr r8
 0x000085e8 <+76>: andeq r0, r1, r12, asr r8
End of assembler dump.
```

cea tech — FROM RESEARCH TO INDUSTRY

INSERTION OF DUMMY INSTRUCTIONS

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

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

    nbIt_Coron++;
}
```

Compilation en –Os:

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

It works!

… but it's fragile!

- Protection contre les fuites par canaux cachés en distance de Hamming
- Fuite sur la valeur *v*, stockée dans un registre ou en mémoire:

```
insn_k
mem <- v
```
→ HD(v,k)

ou:

```
insn_k
reg <- v
```
→ HD(v,k)

Fuite : HD(v,k)

- **Random precharging**: l'affectation est précédée du chargement d'un masque aléatoire m inconnu de l'attaquant:

```
insn_k
mem <- m
mem <- v
```

ou:

```
insn_k
reg <- m
reg <- v
```

Fuite : HD(v,m) = HW(v⊕m)

```
#define SBOX_SIZE   16
uint8_t sbox[SBOX_SIZE];
uint8_t state[SBOX_SIZE];

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

    for(i = 0; i<SBOX_SIZE; i++) {
        state[i] = sbox[state[i]];
    }
}
```

Compilation en –Os:

```
Dump of assembler code for function subBytes:
   0x000084f4 <+0>:  ldr r3, [pc, #28] ; <subBytes+36>
   0x000084f8 <+4>:  ldr r0, [pc, #28] ; <subBytes+40>
   0x000084fc <+8>:  add r2, r3, #16
   0x00008500 <+12>: ldrb r1, [r3, #1] ; r1 <- state[i]
   0x00008504 <+16>: ldrb r1, [r0, r1] ; r1 <- sbox[r1]
   0x00008508 <+20>: strb r1, [r3, #1]! ; leakage hypotl
   0x0000850c <+24>: cmp r3, r2
   0x00008510 <+28>: bne 0x8500 <subBytes+12>
   0x00008514 <+32>: bx lr
   0x00008518 <+36>: andeq r0, r1, r8, lsr r7
   0x0000851c <+40>: andeq r0, r1, r9, asr #14
End of assembler dump.
```

```
#define SBOX_SIZE   16
uint8_t sbox[SBOX_SIZE];
uint8_t state[SBOX_SIZE];

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

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

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

Compilation en –Os:

```
mp of assembler code for function subBytes:
 0x00008524 <+0>:  push {r3, r4, r5, r6, r7, lr}
 0x00008528 <+4>:  ldr r4, [pc, #32] ; <subBytes+44>
 0x0000852c <+8>:  ldr r7, [pc, #32] ; <subBytes+48>
 0x00008530 <+12>: add r5, r4, #16
 0x00008534 <+16>: ldrb r6, [r4, #1] ; tmp <- state[i]
 0x00008538 <+20>: bl 0x83c8 <rand>  ; mask <- rand()
 0x0000853c <+24>: ldrb r3, [r7, r6] ; r3 <- ??
 0x00008540 <+28>: strb r3, [r4, #1]! ; state[i] <- r3
 0x00008544 <+32>: cmp r4, r5
 0x00008548 <+36>: bne 0x8534 <subBytes+16>
 0x0000854c <+40>: pop {r3, r4, r5, r6, r7, pc}
 0x00008550 <+44>: andeq r0, r1, r4, ror r7
 0x00008554 <+48>: andeq r0, r1, r5, lsl #15
d of assembler dump.
```

```c
#define SBOX_SIZE   16
uint8_t sbox[SBOX_SIZE];
uint8_t volatile state[SBOX_SIZE];

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

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

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

Compilation en –Os:

```
Dump of assembler code for function subBytes:
   0x00008524 <+0>:  push {r3, r4, r5, r6, r7, lr}
   0x00008528 <+4>:  ldr r5, [pc, #48] ; <subBytes+60>
   0x0000852c <+8>:  ldr r7, [pc, #48] ; <subBytes+64>
   0x00008530 <+12>: mov r4, #0
   0x00008534 <+16>: ldrb r6, [r5, r4]
   0x00008538 <+20>: bl 0x83c8 <rand>
   0x0000853c <+24>: and r6, r6, #255 ; 0xff
   0x00008540 <+28>: ldrb r3, [r7, r6]
   0x00008544 <+32>: and r0, r0, #15
   0x00008548 <+36>: strb r0, [r5, r4]
   0x0000854c <+40>: strb r3, [r5, r4]
   0x00008550 <+44>: add r4, r4, #1
   0x00008554 <+48>: cmp r4, #16
   0x00008558 <+52>: bne 0x8534 <subBytes+16>
   0x0000855c <+56>: pop {r3, r4, r5, r6, r7, pc}
   0x00008560 <+60>: andeq r0, r1, r5, lsl #15
   0x00008564 <+64>: muleq r1, r5, r7
End of assembler dump.
```

```c
#define SBOX_SIZE   16
uint8_t sbox[SBOX_SIZE];
uint8_t volatile state[SBOX_SIZE];

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

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

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

Compilation en –O1:

```
ump of assembler code for function subBytes:
   0x00008514 <+0>:  push {r3, r4, r5, r6, r7, lr}
   0x00008518 <+4>:  mov r4, #0
   0x0000851c <+8>:  ldr r5, [pc, #44] ; <subBytes+60>
   0x00008520 <+12>: ldr r7, [pc, #44] ; <subBytes+64>
   0x00008524 <+16>: ldrb r6, [r5, r4]
   0x00008528 <+20>: and r6, r6, #255 ; 0xff
   0x0000852c <+24>: bl 0x83c8 <rand>
   0x00008530 <+28>: and r0, r0, #15
   0x00008534 <+32>: strb r0, [r5, r4]
   0x00008538 <+36>: ldrb r3, [r7, r6]
   0x0000853c <+40>: strb r3, [r5, r4]
   0x00008540 <+44>: add r4, r4, #1
   0x00008544 <+48>: cmp r4, #16
   0x00008548 <+52>: bne 0x8524 <subBytes+16>
   0x0000854c <+56>: pop {r3, r4, r5, r6, r7, pc}
   0x00008550 <+60>: andeq r0, r1, r8, lsl #15
   0x00008554 <+64>: muleq r1, r8, r7
End of assembler dump.
```

Bon?? Bah…

Je compile sans

optimisations alors ??

- **Compilation -O0 : placement mémoire systématique de toutes les variables du programme !**
- **Register spilling (> -O0) : la valeur du registre est copiée sur la pile pour libérer l'utilisation du registre**

  => fuite d'information !
  => Insertion de points de vulnérabilité supplémentaires !

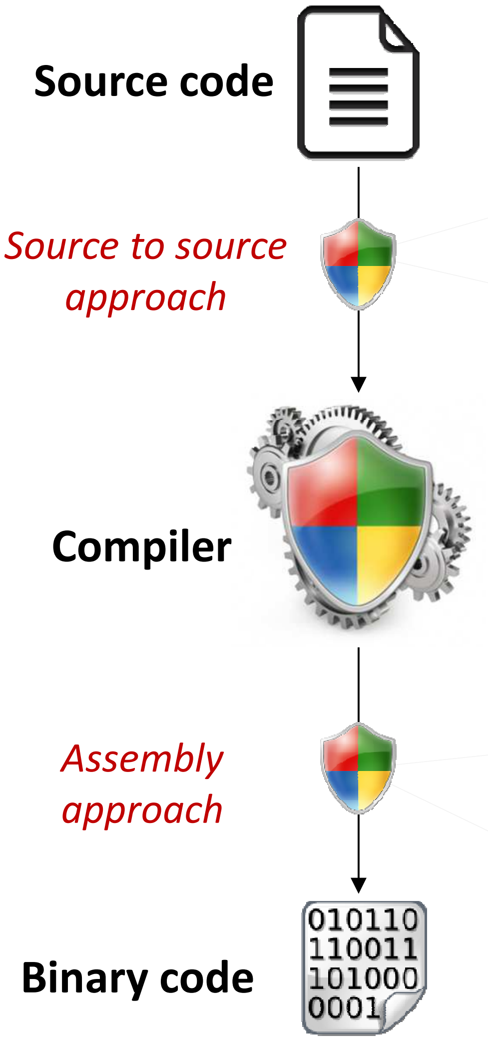
```c
void subBytes(void)
{
    size_t i;
    uint8_t mask, tmp_state;

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

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

```
Dump of assembler code for function subBytes:
0x000084e4 <+0>:  push {r11}  ; (str r11, [sp, #-
0x000084e8 <+4>:  add  r11, sp, #0
0x000084ec <+8>:  sub  sp, sp, #12
0x000084f0 <+12>: mov  r3, #0
0x000084f4 <+16>: str  r3, [r11, #-8]
0x000084f8 <+20>: b    0x8530 <subBytes+76>
0x000084fc <+24>: ldr  r2, [pc, #68] ; <subBytes+
0x00008500 <+28>: ldr  r3, [r11, #-8]
0x00008504 <+32>: add  r3, r2, r3
0x00008508 <+36>: ldrb r3, [r3]
0x0000850c <+40>: ldr  r2, [pc, #56] ; <subBytes+
0x00008510 <+44>: ldrb r2, [r2, r3]
0x00008514 <+48>: ldr  r1, [pc, #44] ; <subBytes+
0x00008518 <+52>: ldr  r3, [r11, #-8]
0x0000851c <+56>: add  r3, r1, r3
0x00008520 <+60>: strb r2, [r3]
0x00008524 <+64>: ldr  r3, [r11, #-8]
0x00008528 <+68>: add  r3, r3, #1
0x0000852c <+72>: str  r3, [r11, #-8]
0x00008530 <+76>: ldr  r3, [r11, #-8]
0x00008534 <+80>: cmp  r3, #15
0x00008538 <+84>: bls  0x84fc <subBytes+24>
0x0000853c <+88>: sub  sp, r11, #0
0x00008540 <+92>: pop  {r11} ; (ldr r11, [sp], #4
0x00008544 <+96>: bx   lr
0x00008548 <+100>:andeq r0, r1, r4, lsl #15
0x0000854c <+104>:muleq r1, r4, r7
```

APPLICATION DE PROTECTIONS PAR COMPILATION (STATIQUE & DYNAMIQUE)

NO MORE –O0,
NO MORE PROGRAMMING IN ASSEMBLY,
NO MORE COOKING RECIPES

APPLICATION AUTOMATISÉE DE PROTECTIONS PAR COMPILATION STATIQUE



✓ Accès à la sémantique du programme (e.g. variable secrète ?)

× Propriétés sécuritaires : pas de garantie de conservation après compilation [Eldib 2014]

Corrolaire : peut donner lieu à des overheads importants

+400% dans [Lalande 2014]

← **Notre approche**

✓ Accès à la sémantique
✓ Accès au code machine
✓ Accès aux optimisations du compilateur
× Mise en œuvre difficile

✓ Application de protections sur le code machine (e.g. duplication d'instructions)

× Reconstruction d'une représentation du programme : complexe au-delà du bloc de base

× Application de la contremesure souvent ad hoc (e.g. [Barenghi 2010])

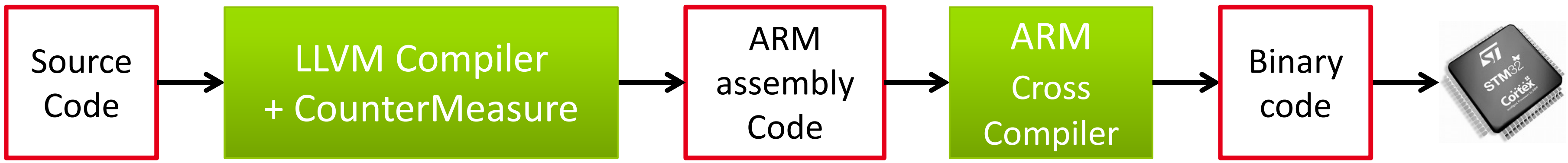
COMPILATION OF A COUNTERMEASURE AGAINST INSTRUCTION SKIP FAULT ATTACKS

- 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

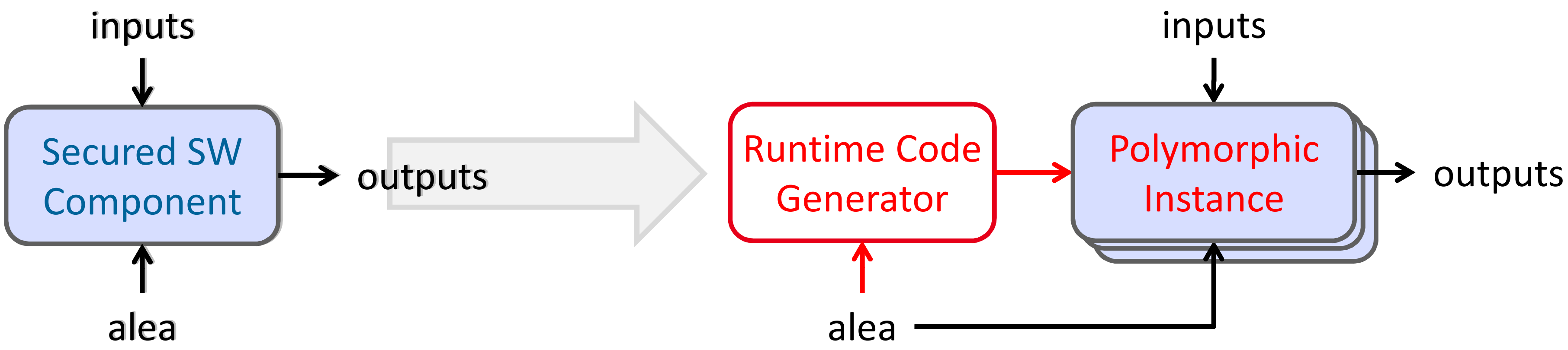


- Experimental results for AES

	Opt. level	Unprotected		Protected		Overhead		Moro et al [12]	
		cycles	size	cycles	size	cycles	size	cycles	size
Moro et al.'s AES	-O0	8439	13472	16713	17344	×1.98	×1.28	×2.14	×3.02
	-Os	13409	12448	25432	14992	×1.89	×1.20		
	-O1	16828	11792	32973	13712	×1.96	×1.16		
	-O2	14407	11552	24703	13248	×1.71	×1.15		
	-O3	14407	11552	24703	13248	×1.71	×1.15		
MiBench AES	-O0	1890	66332	3817	83676	×2.02	×1.26	×2.86	×2.90
	-Os	1908	66988	3355	79292	×1.75	×1.18		
	-O1	1142	64604	2188	68988	×1.96	×1.16		
	-O2	1142	60092	2188	69452	×1.96	×1.16		
	-O3	1908	67644	3355	79948	×1.76	×1.18		

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

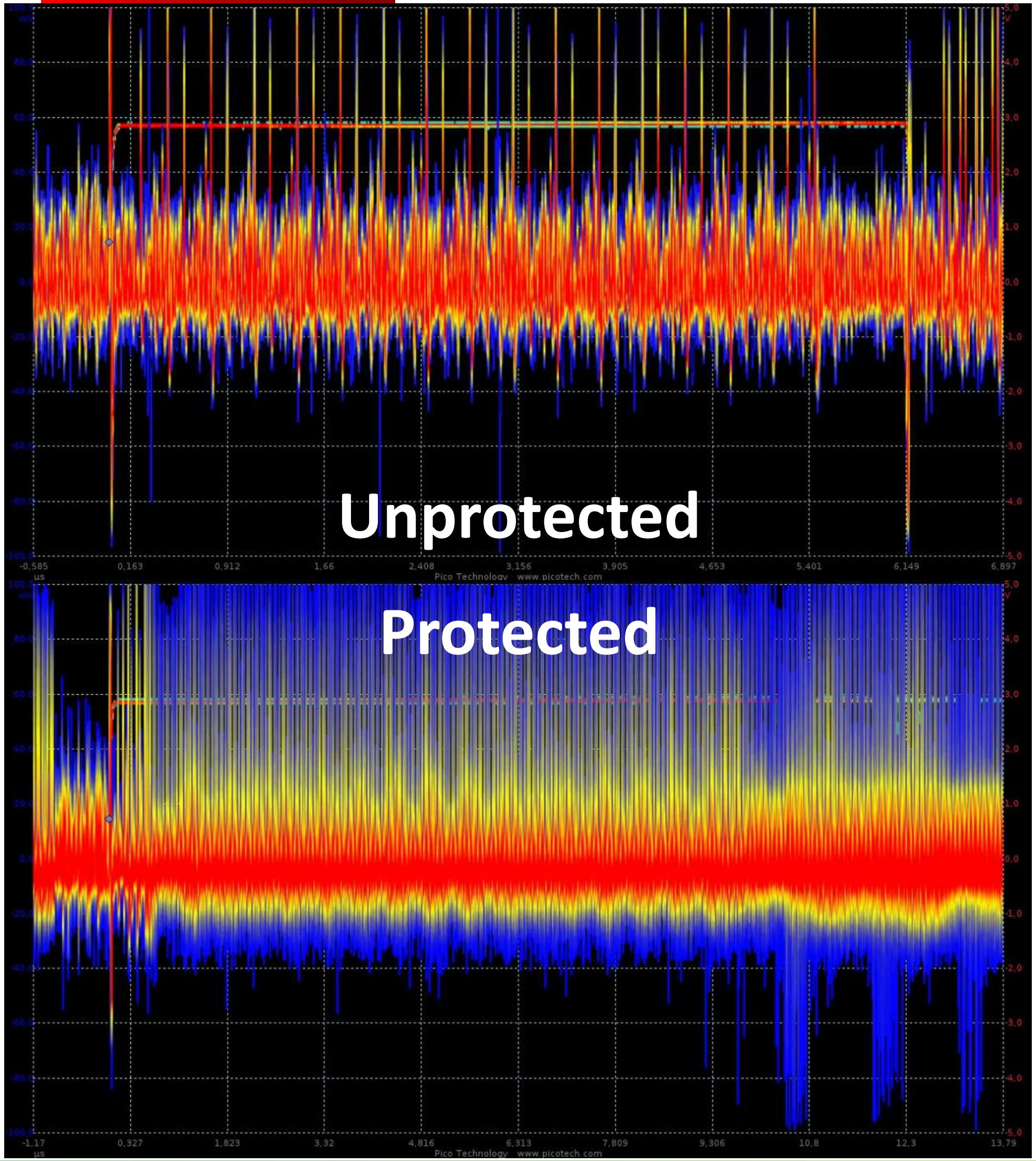
- **Regularly changing the behavior of a (secured) component, at runtime, while maintaining unchanged its functional properties, with runtime code generation**

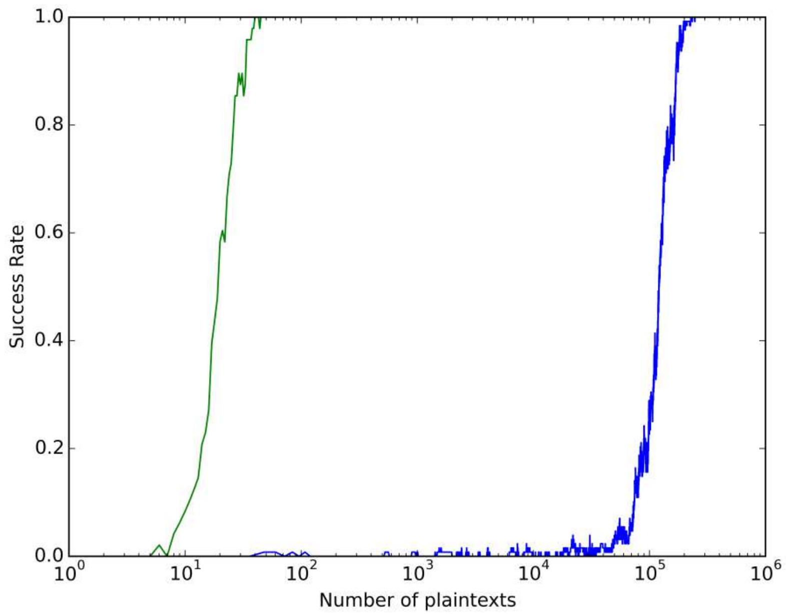
- **Protection against reverse engineering of SW**
  - the secured code is not available before runtime
  - the secured code regularly changes its form (code generation interval ω)

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

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

ILLUSTRATION ON AES





Compilation *and* cyber-security in embedded systems

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