Why there is no silver bullet to solve our security issues

Ronan Lashermes [1]

[1]INRIA/SED&LHS

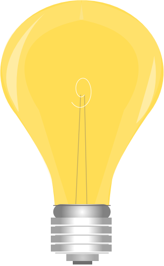
November 5th, 2019

ENS, Rennes

Section 1

Introduction

A small enigma



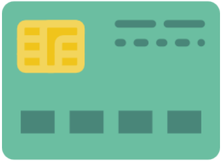
Section 2

How software can go wrong

Designing a PIN verification

Your mission

In order to enter a very secure room, the visitor must insert her personal **smartcard** and enter the corresponding unique valid **PIN**. If the card and the PIN match, the visitor can enter. If not, she has 3 **attempts** or she faces jail time.



Initial proposal, spot the vulnerabilities

```c
#define PIN_SIZE 4
#define MAX_TRY 3
typedef unsigned char BYTE;

int try_counter = MAX_TRY;
BYTE pin_correct[PIN_SIZE];
BYTE pin_candidat[PIN_SIZE];

void enter_pin() {
  printf("A␣PIN␣has␣%d␣digits.", PIN_SIZE);
  printf("Please␣enter␣PIN:␣");
  scanf("%s", pin_candidat);
}

bool compare_arrays( // memcmp
  BYTE *arr1,
  BYTE *arr2) {
  for(int i = 0; i < PIN_SIZE; i++) {
    if(arr1[i]!=arr2[i]) return false; }
  return true;
}

void verify_pin(
  BYTE *pin_candidat,
  BYTE *pin_correct) {
  if(compare_arrays(
    pin_candidat,
    pin_correct) == true) { // PIN ok

    try_counter = MAX_TRY;
    authenticate();
  } else { // PIN not correct
    try_counter -= 1;
    if(try_counter <= 0) {
      kill(); }
    else {
      incorrect(); }
  }
}
```

Buffer overflow

```c
int try_counter = MAX_TRY;
BYTE pin_correct[PIN_SIZE];
BYTE pin_candidat[PIN_SIZE];

void enter_pin() {
  printf("A␣PIN␣has␣%d␣digits.", PIN_SIZE);
  printf("Please␣enter␣PIN:␣");
  scanf("%s", pin_candidat);
}
```

Buffer overflow

```
int try_counter = MAX_TRY;
BYTE pin_correct[PIN_SIZE];
BYTE pin_candidat[PIN_SIZE];

void enter_pin() {
  printf("A␣PIN␣has␣%d␣digits.", PIN_SIZE);
  printf("Please␣enter␣PIN:␣");
  scanf("%s", pin_candidat);
}
```

Exploit

```
> A PIN has 4 digits.
> Please enter your PIN: aaaaaaaa
> PIN ok.
```

Buffer overflow

```
int try_counter = MAX_TRY;
BYTE pin_correct[PIN_SIZE];
BYTE pin_candidat[PIN_SIZE];

void enter_pin() {
    printf("A␣PIN␣has␣%d␣digits.", PIN_SIZE);
    printf("Please␣enter␣PIN:␣");
    scanf("%s", pin_candidat);
}
```

Exploit

```
> A PIN has 4 digits.
> Please enter your PIN: aaaaaaaa
> PIN ok.
```

Solutions

Google gives you 6950000 results. Some examples:

- Canaries.
- Do not use scanf, better variants exist.
- ...

My take: this is a language design problem

Figure: Door picture from uxdesign.cc

Securely programming in C is like playing with a loaded gun. Of course it can end well, but you should probably not be doing it.

Proof-oriented software[1]

Formal methods

Being able to write bug-free programs is one of the goal of formal languages such as Coq, Lean, and numerous others. They use the Curry-Howard correspondance to reduce writing a correct program to proving a mathematical proof.

But

- A mathematical proof is something abstract while an executed program has a physical existence. A point already made in 1988 in "Program verification: the very idea" by J.H. Fetze.
- There is a reason even mathematicians do not use these tools pervasively (cf the talks and writing of Kevin Buzzard).

[1] I made up this expression, don't google it

Against the clock

```
bool compare_arrays( // memcmp
  BYTE *arr1,
  BYTE *arr2) {
  for(int i = 0; i < PIN_SIZE; i++) {
    if(arr1[i]!=arr2[i]) return false; }
  return true;
}
```

Timing leakage

Supposing illimited tries, measure the duration of the *compare_arrays* function.

```
> Please enter your PIN: 1111 => 3us
> Please enter your PIN: 2222 => 3us
> Please enter your PIN: 3333 => 3us
> Please enter your PIN: 4444 => 4us
> Please enter your PIN: 4111 => 4us
> Please enter your PIN: 4211 => 6us
> Please enter your PIN: 4212 => 6us
> Please enter your PIN: 4213 => 7us
> PIN ok.
```

Constant-time implementations

```
bool compare_arrays( // memcmp
  BYTE *arr1,
  BYTE *arr2) {
  BYTE diff = 0;
  for(int i = 0 ; i < PIN_SIZE ; i++)
    diff &= arr1[i] ^ arr2[i];
  return diff == 0;
}
```

Constant-time implementations

```
bool compare_arrays( // memcmp
  BYTE *arr1,
  BYTE *arr2) {
  BYTE diff = 0;
  for(int i = 0 ; i < PIN_SIZE ; i++)
    diff &= arr1[i] ^ arr2[i];
  return diff == 0;
}
```

What guarantees constant-time execution ?

Figure: Going down the rabbit hole

Section 3

How hardware can go wrong

Cache-timing attacks

What variable(s) leaks ? (x, y, arr)

```
int x = arr[y];
```

Cache-timing attacks

What variable(s) leaks ? (x, y, arr)

```c
int x = arr[y];
```

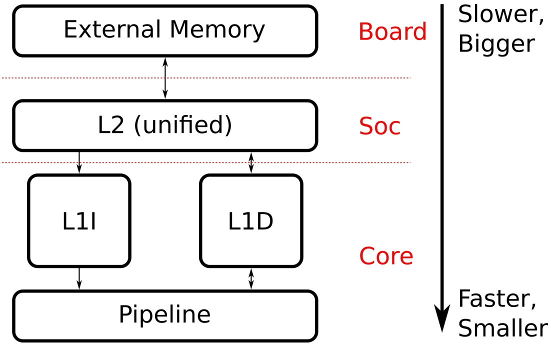


Figure: Memory hierarchy

Meltdown and Spectre

Micro-architecture

The implementation of the instruction set architecture (ISA).

Meltdown and Spectre

Micro-architecture

The implementation of the instruction set architecture (ISA).

Attacks' principle

Read somewhere in forbidden memory, using transient instructions. Use cache-timing to recover the result.

Meltdown and Spectre

Micro-architecture

The implementation of the instruction set architecture (ISA).

Attacks' principle

Read somewhere in forbidden memory, using transient instructions. Use cache-timing to recover the result.

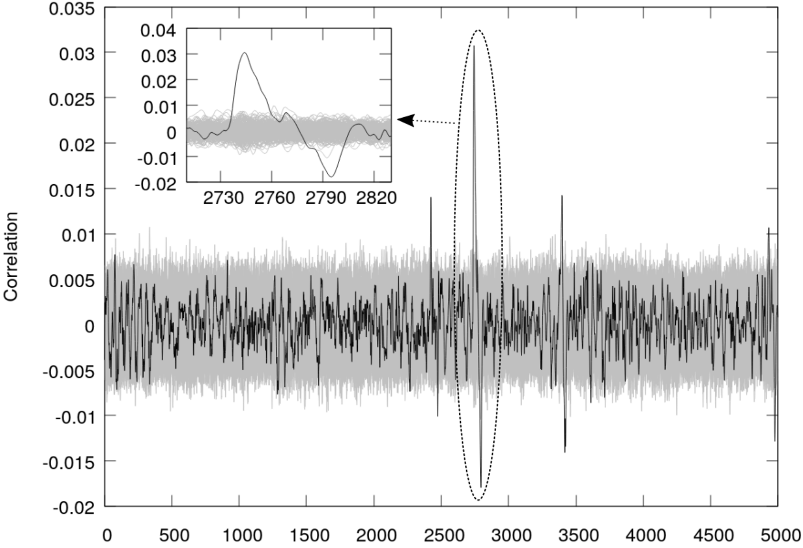
Recover kernel data from application (Spectre)

```
if(x < array1_size)
  y = array2[array1[x]*4096];
```

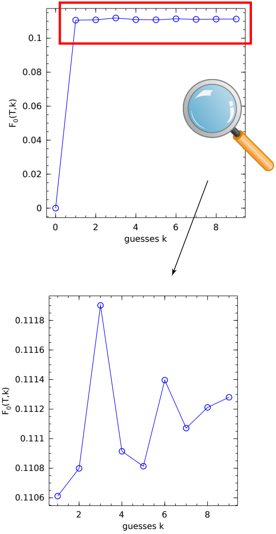
Attacker controls `x` and wants to read arbitrarily in memory.

Side-channel analysis

Correlation Power Analysis



SCA against PIN verification



Template attack

- Learn the leakage on a controlled device.
- Measure on the target device and compare (Mahalanobis distance).

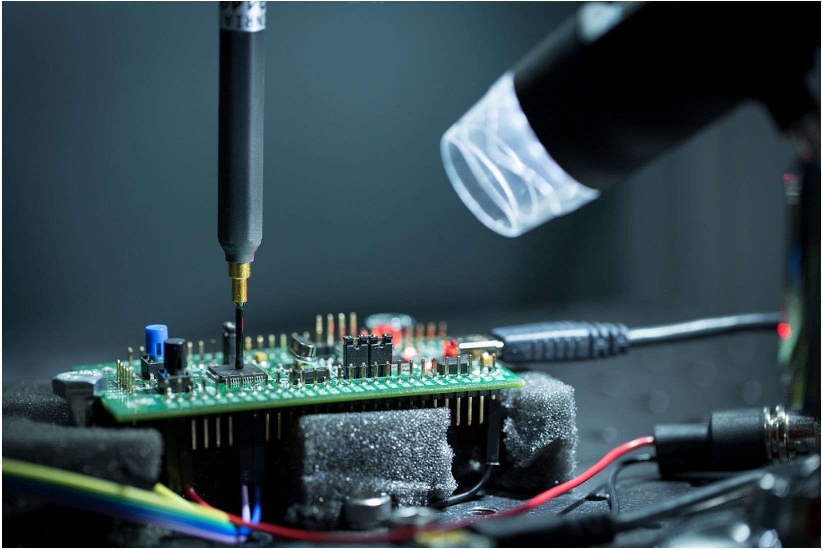
Protecting against SCA

Masking

We want to protect s when computing $o = f(s)$, with f linear. Generate a truly random value r, then **mask** the secret: $s \oplus r$.
Finally, you can compute o without depending on the secret value:

$$o = f(r) \oplus f(s \oplus r).$$

Fault injection attacks



Fault activated backdoor

```c
void blink_wait()
{
  unsigned int wait_for = 3758874636;
  unsigned int counter;
  for(counter = 0; counter < wait_for; counter += 8000000);
}
```

Fault activated backdoor

```c
void blink_wait()
{
  unsigned int wait_for = 3758874636;
  unsigned int counter;
  for(counter = 0; counter < wait_for; counter += 8000000);
}
```

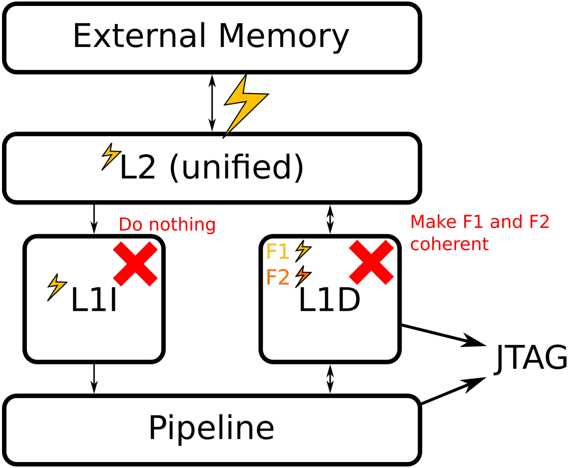
```
08000598 <blink_wait>:
push     {r7, lr}
sub      sp, #8
add      r7, sp, #0
ldr      r3, [pc, #44]    ; (80005cc <blink_wait+0x34>)
...
adds     r7, #8
mov      sp, r7
pop      {r7, pc}
.word    0xe00be00c ; @80005cc, 0xe00be00c = 3758874636
```

Fault activated backdoor

```c
void blink_wait()
{
  unsigned int wait_for = 3758874636;
  unsigned int counter;
  for(counter = 0; counter < wait_for; counter += 8000000);
}
```

```
08000598 <blink_wait>:
push    {r7, lr}
sub     sp, #8
add     r7, sp, #0
ldr     r3, [pc, #44]    ; (80005cc <blink_wait+0x34>)
...
adds    r7, #8
mov     sp, r7
nop ; pop        {r7, pc}
b other_verif ;.word    (0xe00b)e00c
b other_verif ;.word    0xe00b(e00c)
```

Faults on the memory hierarchy



FIA against PIN verification

```
void verify_pin(BYTE *pin_candidat, BYTE *pin_correct) {
  if(compare_arrays(pin_candidat, pin_correct) == true) { // PIN ok
    try_counter = MAX_TRY;
    authenticate();
  }
  else { // PIN not correct
    try_counter -= 1;
    if (try_counter <= 0) {
      kill();
    }
    else {
      incorrect();
    }
  }
}
```

Protecting PIN verification against fault attacks

Duplicate all the things !

```
void verify_pin(BYTE *pin_candidat, BYTE *pin_correct) {
  if(compare_arrays(pin_candidat, pin_correct) == true) { // PIN ok 1
        if (compare_arrays(pin_candidat, pin_correct) == true) { //PIN ok 2
            try_counter = MAX_TRY;
            authenticate();
        }
  }
  else { // PIN not correct
    try_counter -= 1;
    if (try_counter <= 0) {
      kill();
    }
    else {
      incorrect();
    }
  }
}
```

Protecting PIN verification against fault attacks

Duplicate all the things !

```
void verify_pin(BYTE *pin_candidat, BYTE *pin_correct) {
  if(compare_arrays(pin_candidat, pin_correct) == true) { // PIN ok 1
        if (compare_arrays(pin_candidat, pin_correct) == true) { //PIN ok 2
            try_counter = MAX_TRY;
            authenticate();
        }
  }
  else { // PIN not correct
    try_counter -= 1;
    if (try_counter <= 0) {
      kill();
    }
    else {
      incorrect();
    }
  }
}
```

Actually really painful. What if the attacker can inject 2 faults ?

Section 4

Hardened PIN verification

Implementation countermeasures

- Input sanitization.
- Formal methods to detect implementations not meeting specifications.
- Constant-time implementation.
- Redundancy against FIA.
- Masking against SCA.

If one of these countermeasures is too weak, your implementation is unsecure. (The weakest link rule)

Implementation countermeasures

- Input sanitization.
- Formal methods to detect implementations not meeting specifications.
- Constant-time implementation.
- Redundancy against FIA.
- Masking against SCA.

If one of these countermeasures is too weak, your implementation is unsecure. (The weakest link rule)

Can we do better ?

Changing the protocol

Cryptography to the rescue

It would be better if the secret were not present inside the chip, but verification still possible.

Solution

Generate *pin*, *key* when enrolling the smartcard. Store

$$(key, commited = HMAC(key, pin)).$$

Now to verify *pin_try*, the smartcard computes

$$hash\_try = HMAC(key, pin\_try).$$

And we compare *commited* and *hash_try*.

Advantages

- The secret can be forgotten, no *pin* value to recover.
- Comparison do not require to be secure.
- The embedded *key* prevents to compare leakage from two devices.

But fault attacks are still possible.

Section 5

Conclusion

Leaking interfaces

Interfaces between abstraction do not define security properties. As a result, they leak information.

You cannot implement a secure functionality without taking into account all abstraction layers (protocol, algorithms, software, hardware).

Thank you!

Any questions?