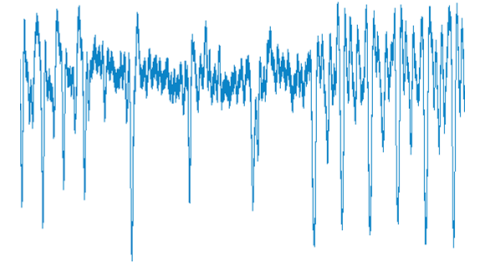


Implementation Attacks

David Oswald



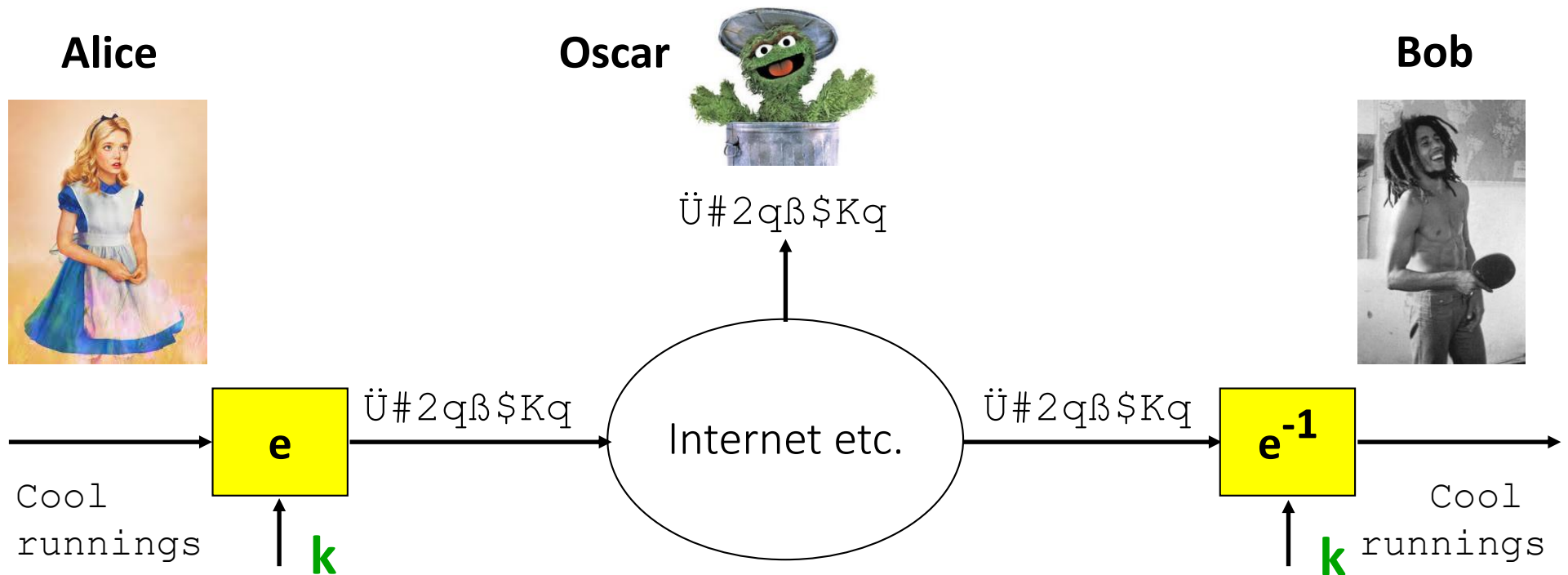
Classical security assumption:

Attacker knows x , $e_k(x)$, and $e()$ but **not** k

Problem for adversary

Modern crypto often secure against classical attacks:

- AES, XSalsa20, ...
- RSA, ECC, ...



Principle of Side-Channel Analysis (here: listen to Sound)

A Bank Robbery



Principle of Side-Channel Analysis

The world is changing...



Principle of Side-Channel Analysis

(now: measure the power consumption / run-time)


The world is changing...



...the tools are, too.



Possible Side Channels

- General goal: Recover secrets (keys and more)
- **Runtime** (timing attacks) 

A timing attack on PIN checks

```
bool check_pin(int pin_entered[4]) ≈ memcmp()  
{  
    for(int i = 0; i < 4; i++) {  
        if(pin_stored[i] != pin_entered[i]) {  
            return false;  
        }  
    }  
    return true;  
}
```

Recovering the PIN

Entered PIN	Runtime (μ s)
0000	134
1000	133
2000	166
3000	129
4000	133
5000	132
6000	134
7000	128
8000	136
9000	129

Recovering the PIN

Entered PIN	Runtime (μ s)
0000	134
1000	133
2000	166
3000	129
4000	133
5000	132
6000	134
7000	128
8000	136
9000	129

Recovering the PIN

Entered PIN	Runtime (μ s)
2000	165
2100	160
2200	166
2300	163
2400	199
2500	159
2600	158
2700	167
2800	166
2900	165

Recovering the PIN

Entered PIN	Runtime (μ s)
2000	165
2100	160
2200	166
2300	163
2400	199
2500	159
2600	158
2700	167
2800	166
2900	165

Recovering the PIN

Entered PIN	Runtime (μ s)
2400	198
2410	201
2420	197
2430	197
2440	198
2450	199
2460	228
2470	195
2480	204
2490	197

Recovering the PIN

Entered PIN	Runtime (μ s)
2400	198
2410	201
2420	197
2430	197
2440	198
2450	199
2460	228
2470	195
2480	204
2490	197

Recovering the PIN

Entered PIN	Result
2460	false
2461	false
2462	false
2463	false
2464	false
2465	false
2466	false
2467	false
2468	true
2469	false

Recovering the PIN

Entered PIN	Result
2460	false
2461	false
2462	false
2463	false
2464	false
2465	false
2466	false
2467	false
2468	true
2469	false

Live-Demo

“Anything that can go wrong,
will go wrong”

A timing attack

```
david@timing_attack$ make clean
rm *.o
david@timing_attack$
```

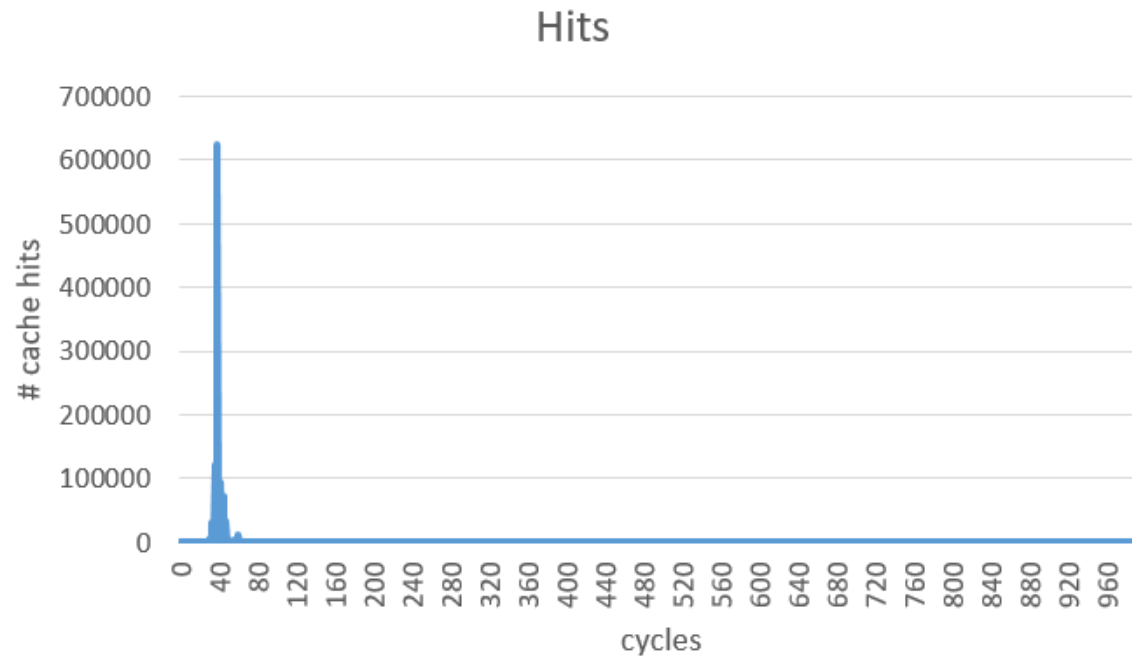
```
File Edit Search View Encoding Language Settings Macro Run TextFX Plugins Window ?
main.c
7  #define USE_RDTSCP 1
8
9  static uint64_t rdtsc()
10 {
11     uint64_t a, d;
12     __asm volatile("mfence");
13     #if USE_RDTSCP
14         __asm volatile("rdtscp" : "=a"(a), "=d"(d) : "rcx");
15     #else
16         __asm volatile("rdtsc" : "=a"(a), "=d"(d));
17     #endif
18     a = (d << 32) | a;
19     __asm volatile("mfence");
20     return a;
21 }
22
23 static int check_pin(int pin_entered[4])
24 {
25     int i = 0;
26
27     for(i = 0; i < 4; i++)
28     {
29         if(pin_stored[i] != pin_entered[i])
30         {
31             return -1;
32         }
33     }
34
35     return 0;
36 }
37
38 //#define PRINT_RESULTS
39
40 int main(void)
41 {
42     // Average each attempt REPEAT times
43     const size_t REPEAT = 1000;
44
45     // Outlier filtering
46     const double THRESHOLD = 220;
47
48     int pin_entered[4] = {0x0, 0x0, 0x0, 0x0};
```

Possible Side Channels

- General goal: Recover secrets (keys and more)
- Runtime (timing attacks)
- Cache timing (used e.g. in Meltdown/Spectre)

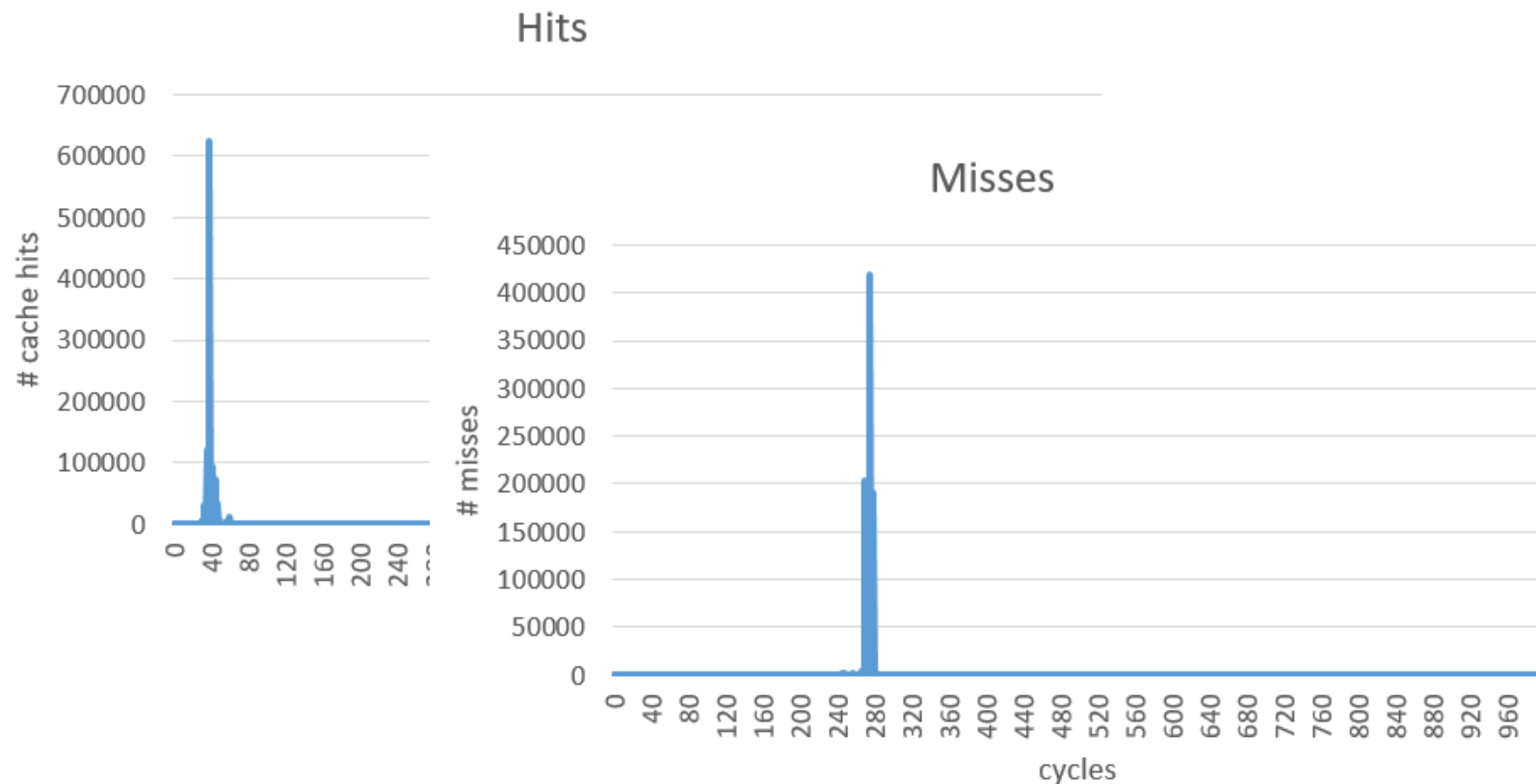
Cache vs. memory

- The cache stores recently used data
- Fetching from cache (“hit”) is faster than from RAM (“miss”)



Cache vs. memory

- The cache stores recently used data
- Fetching from cache (“hit”) is faster than from RAM (“miss”)



Flush+reload

- Flush the cache
- Run the victim
- Measure access times to victim memory

Live-Demo

“Anything that can go wrong,
will go wrong”

Flush and reload

david@cache_timing\$

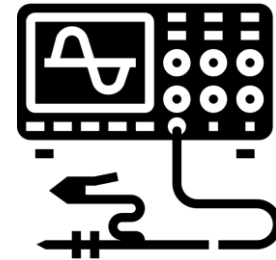
```
File Edit Search View Encoding Language Settings Macro Run TextFX Plugins Window ?
Makefile spy.c
1  #include <emmintrin.h>
2  #include <x86intrin.h>
3  #include <stdio.h>
4  #include <stdint.h>
5
6  // Based on https://seedsecuritylabs.org/Labs\_16.04/System/Meltdown\_Attack/
7
8  #define ARRAY_SIZE 10
9  #define THRESHOLD 200
10
11 uint8_t array[ARRAY_SIZE * 4096];
12
13 int main(int argc, const char **argv)
14 {
15     int junk = 0;
16     register uint64_t time1, time2;
17     volatile uint8_t *addr;
18     int i;
19
20     // Initialize the array
21     for(i = 0; i < ARRAY_SIZE; i++)
22     {
23         array[i*4096] = 1;
24     }
25
26     // FLUSH the array from the CPU cache
27     for(i = 0; i < ARRAY_SIZE; i++)
28     {
29         _mm_clflush(&array[i*4096]);
30     }
31
32     // Access some of the array items - this is the victim code
33     array[3 * 4096] = 100;
34     array[7 * 4096] = 200;
35     // End of "victim"
36
37     // Probe
38     for(i = 0; i < ARRAY_SIZE; i++)
39     {
40         addr = &array[i * 4096];
41
42         time1 = rdtscp(&junk);
```

Flush+reload

- Flush the cache
- Run the victim
- Measure access times to victim memory
- A building block of Spectre/Meltdown
- There are other variants (e.g. Prime+Probe and Flush+Flush)
- Further reading:
https://github.com/IAIK/cache_template_attacks
https://seedsecuritylabs.org/Labs_16.04/System/Meltdown_Attack/

Possible Side Channels

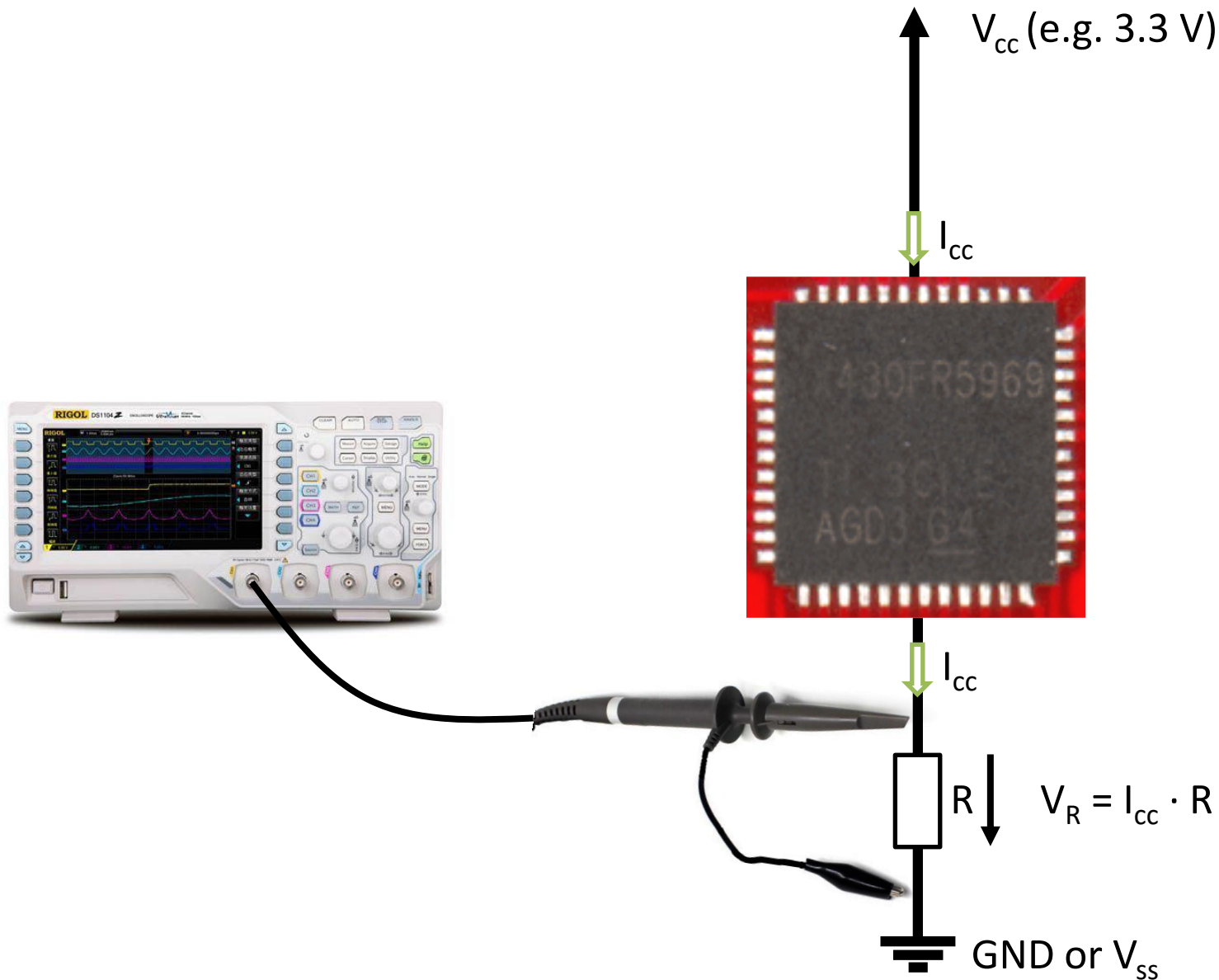
- General goal: Recover secrets (keys and more)
- Runtime (timing attacks)
- Cache timing (used e.g. in Meltdown/Spectre)
- **Power analysis** – let's start with Simple Power Analysis (SPA)



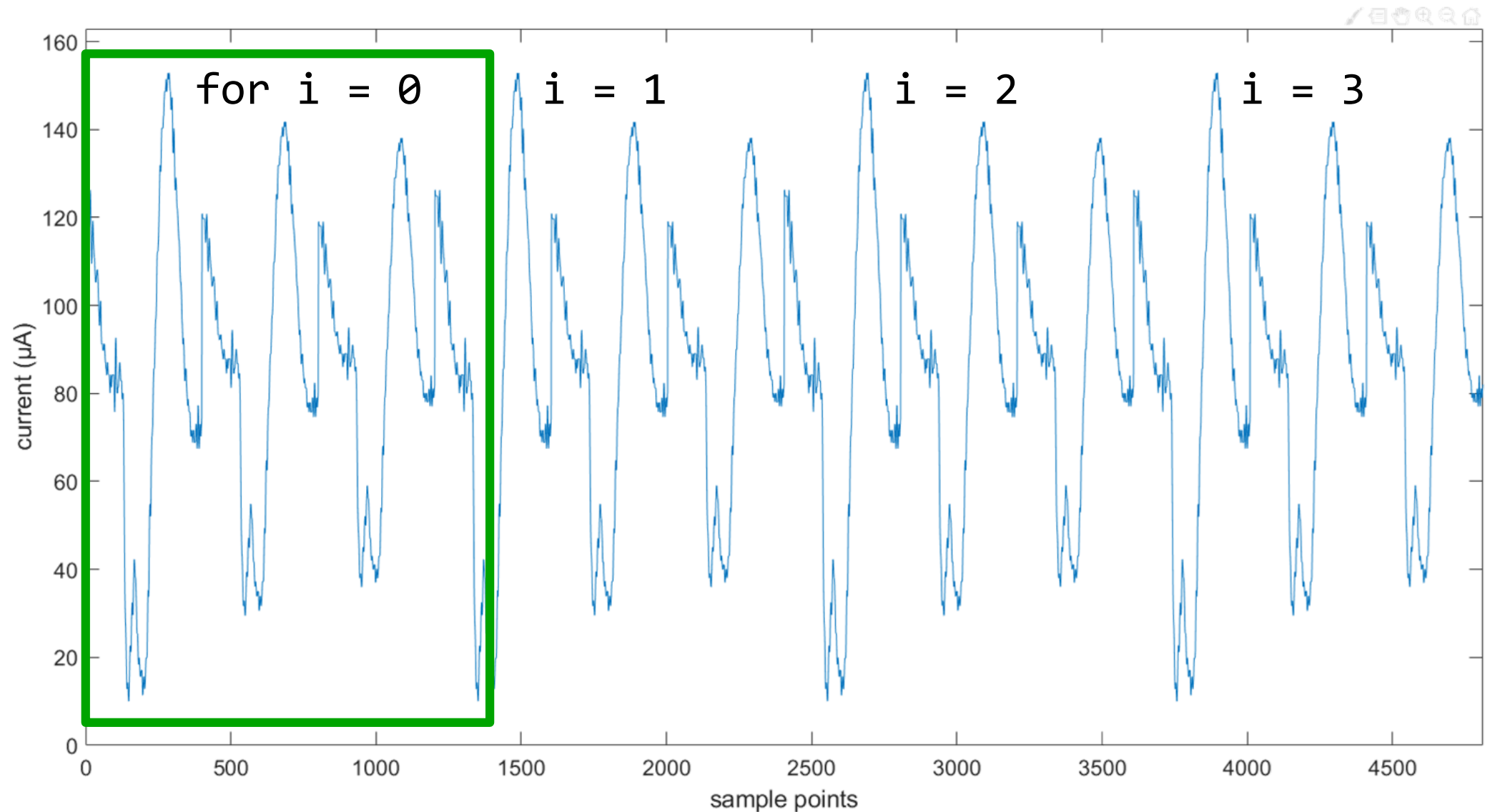
A power analysis attack on PIN checks

```
bool check_pin(int pin_entered[4])
{
    bool flag = true;
    for(int i = 0; i < 4; i++) {
        if(pin_stored[i] != pin_entered[i]) {
            flag = false;
        }
    }
    return flag;
}
```

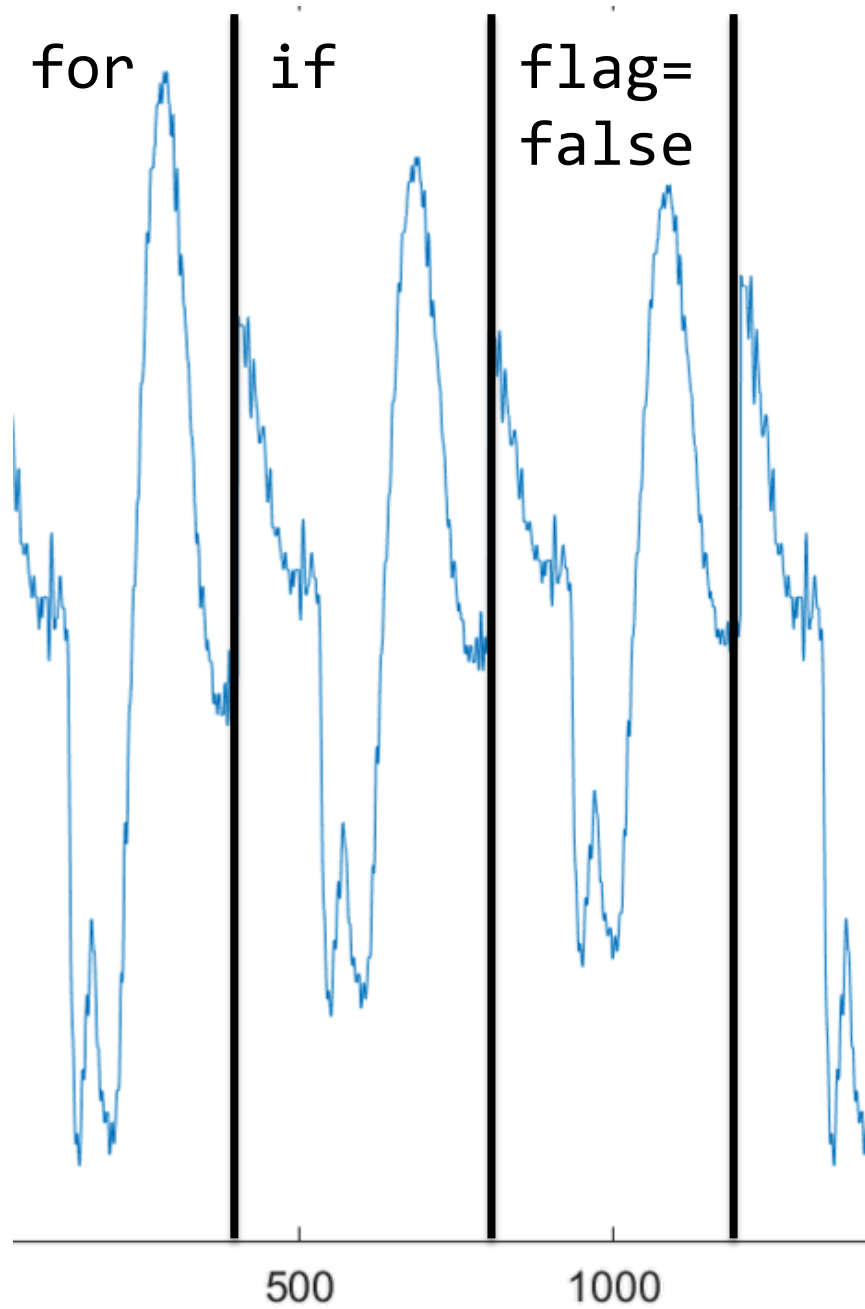

Measuring power consumption



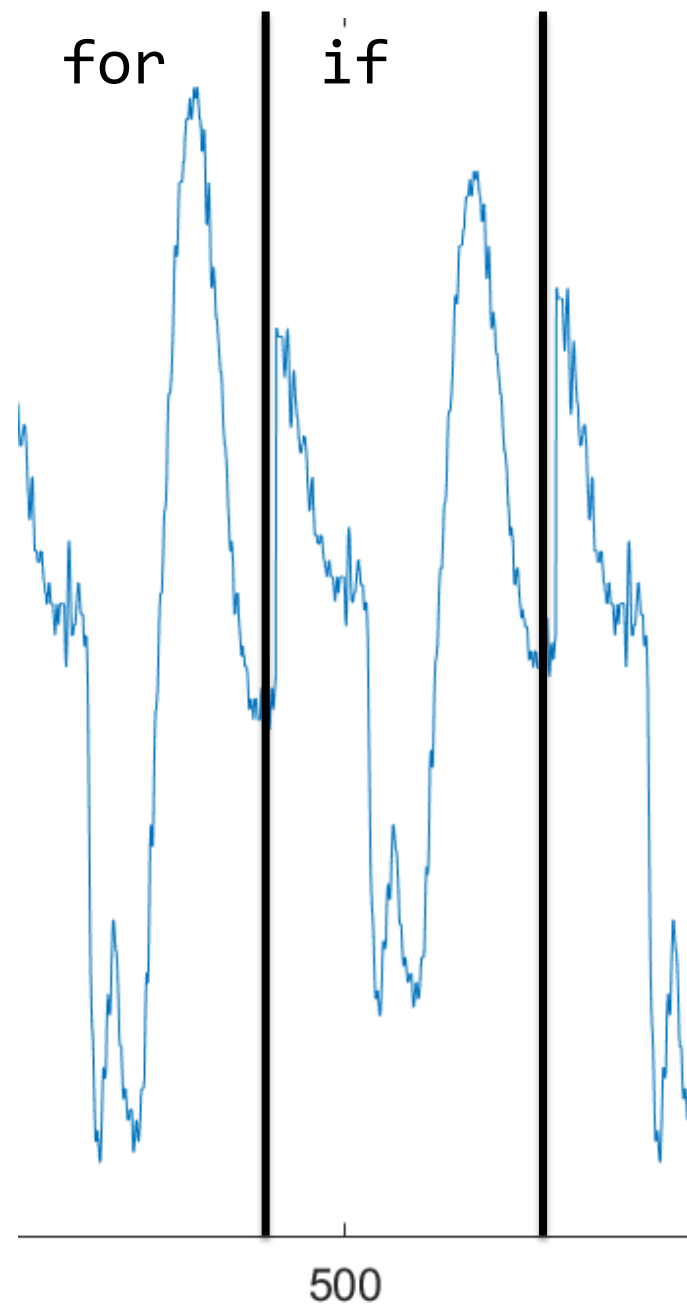
Simple Power Analysis of PIN check: 0000



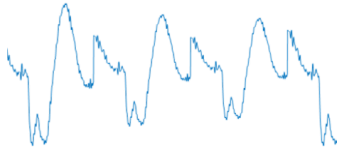
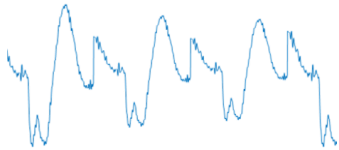
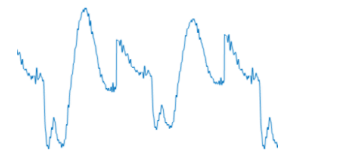
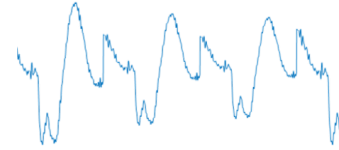
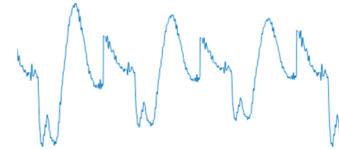
Incorrect digit



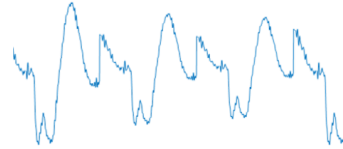
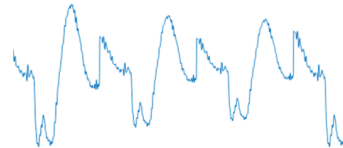
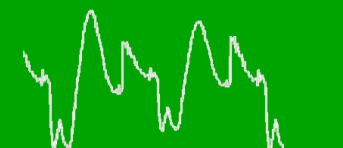
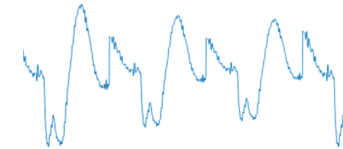
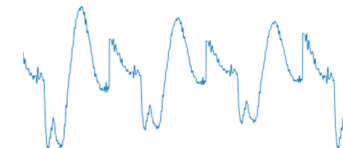
Correct digit



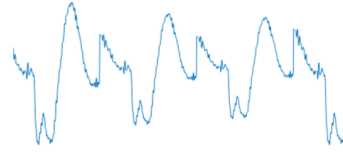
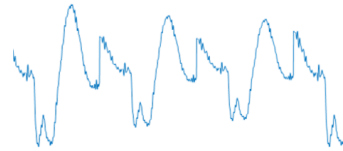
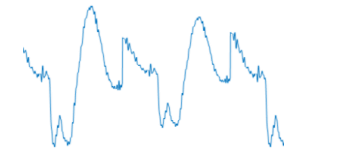
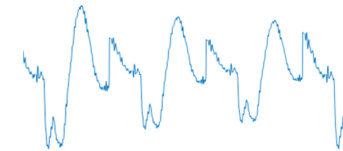
Recovering the PIN with SPA

Entered PIN	Trace ($i = 0$)
0000	
1000	
2000	
...	...
8000	
9000	

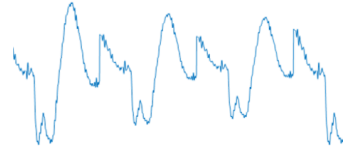
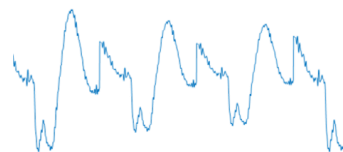
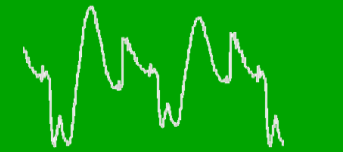
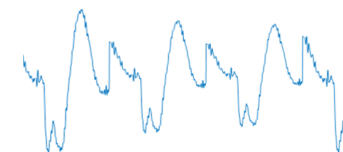
Recovering the PIN with SPA

Entered PIN	Trace ($i = 0$)
0000	
1000	
2000	
...	...
8000	
9000	

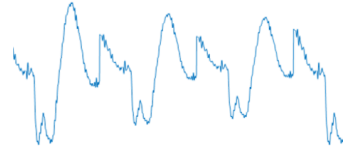
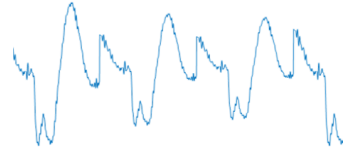
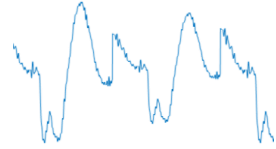
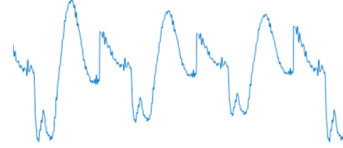
Recovering the PIN with SPA

Entered PIN	Trace (i = 1)
2000	
2100	
...	
2400	
...	...
2900	

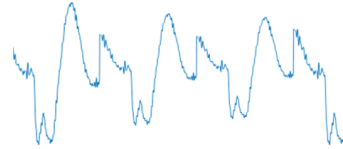
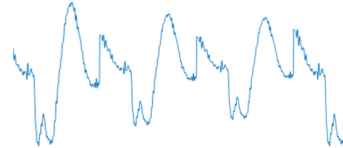
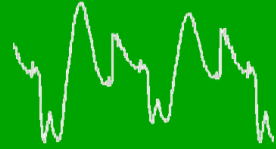
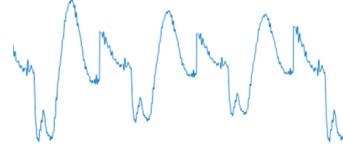
Recovering the PIN with SPA

Entered PIN	Trace (i = 1)
2000	
2100	
...	
2400	
...	...
2900	

Recovering the PIN with SPA

Entered PIN	Trace (i = 2)
2400	
2410	
...	
2460	
...	...
2490	

Recovering the PIN with SPA

Entered PIN	Trace (i = 2)
2400	
2410	
...	
2460	
...	...
2490	





Recovering the PIN with SPA

Entered PIN	Result
2460	false
2461	false
2462	false
...	...
2468	true
2469	false

Recovering the PIN with SPA

Entered PIN	Result
2460	false
2461	false
2462	false
...	...
2468	true
2469	false

Other possible side channels (selection)

- Temperature 
- Sound 
- Electro-magnetic emanations 
- Photonic emissions 
- ...

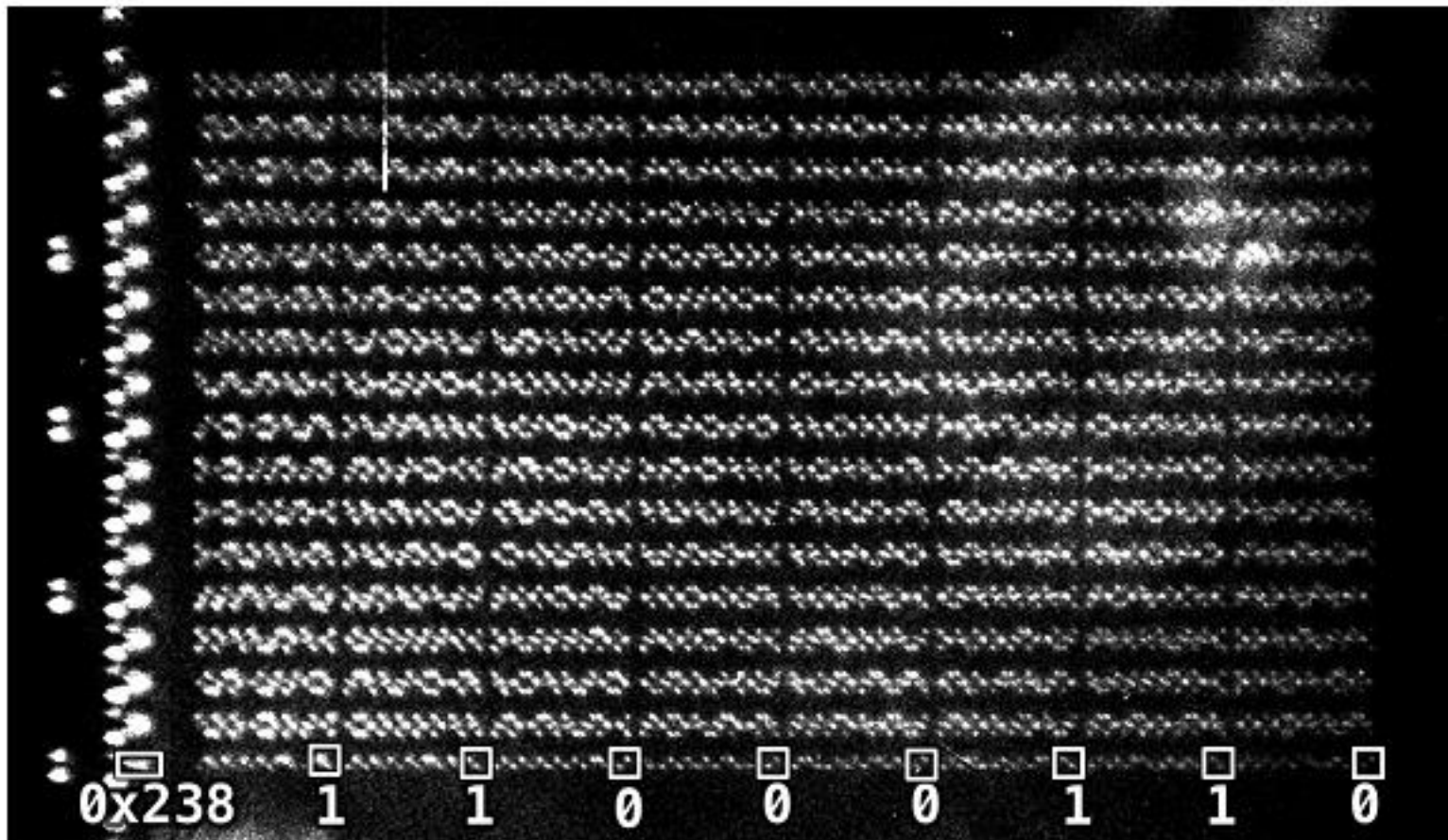


Fig. 3. Optical emission image of the S-Box in memory. The 256 bytes of the S-Box are located from 0x23F to 0x33E, see Table 3 in the appendix. The address 0x23F is the seventh byte of the 0x238 SRAM line, i.e. the S-Box has an offset of 7 bytes. The emissions of the row drivers are clearly visible to the left of the memory bank. The image allows direct readout of the bit-values of the stored data. The first byte for example, as shown in the overlay, corresponds to $01100011_2 = 63_{16}$, the first value of the AES S-Box.

Some countermeasures

- (Cache) timing
 - No secret-dependent branches
 - No secret-indexed memory accesses
 - For example: use “bitslicing”
- Power analysis
 - “Balance” the power consumption (hard)
 - Generate noise (can be overcome with averaging)
 - Randomize timing (can be overcome with averaging)
 - Masking schemes (randomize internal values)

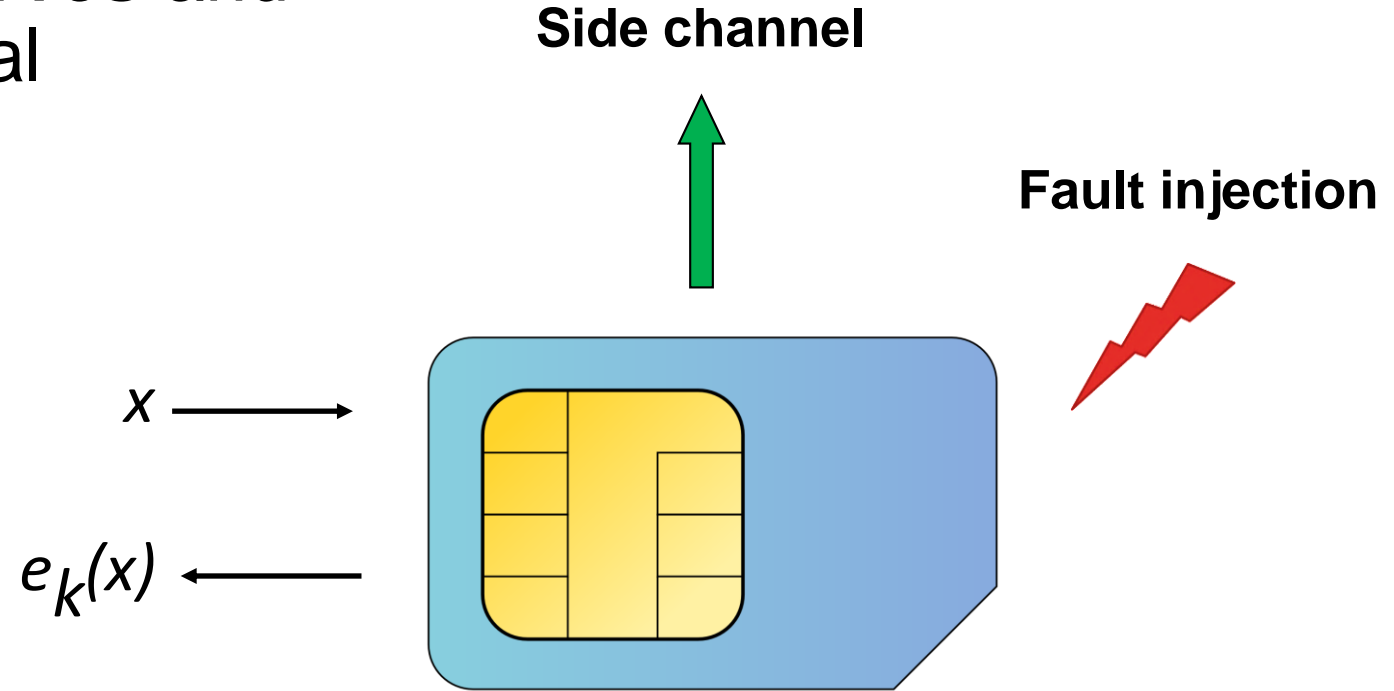
Take-home messages

- Side-channel attacks use *unintended* side effects
- *Independent* of the *mathematical security* of cryptographic algorithms
- Relevant for both small embedded devices (smartcards) and large PC-grade CPUs
- There is also a related class of attacks: fault injection



Embedded reality:

Adversary **observes** and **controls** physical environment



Intuition: “Faulty output $\bar{e}_k(x)$ leaks information on k ”

2. Control: fault injection

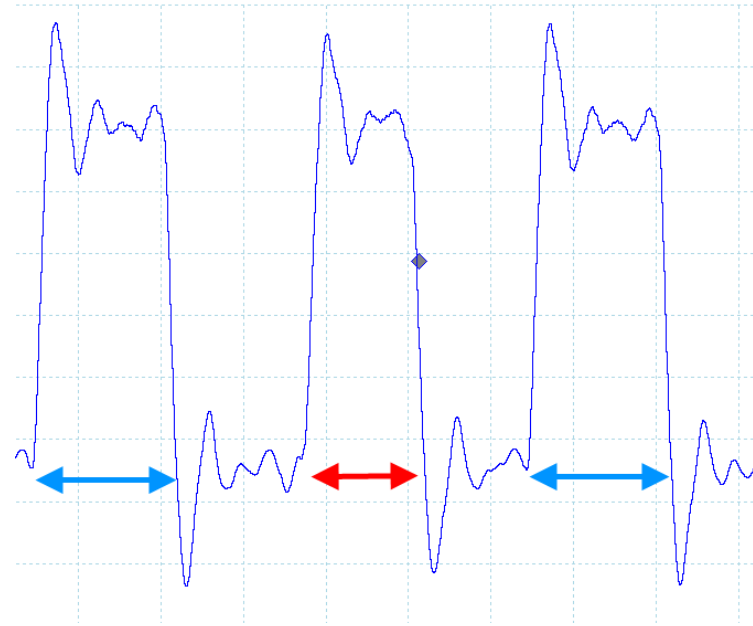
- (Buffer overflows)
- Power/clock glitch
- (Laser) light
- ...

Injecting faults (examples)

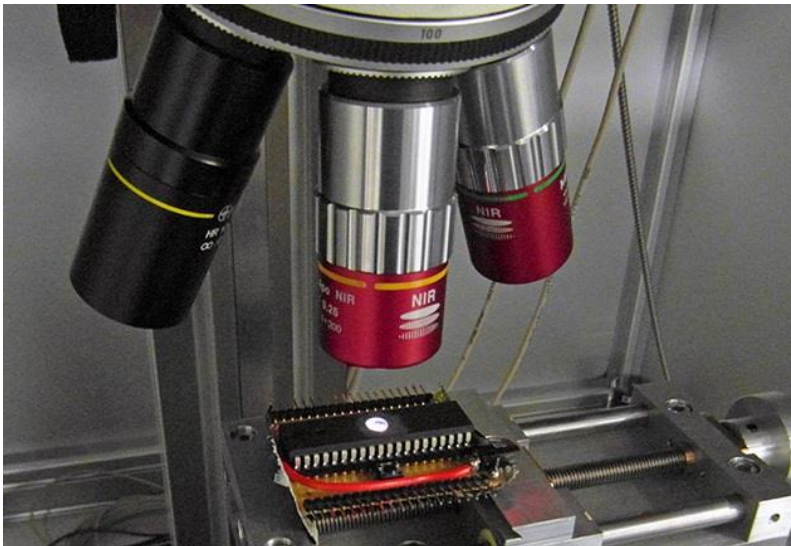
power



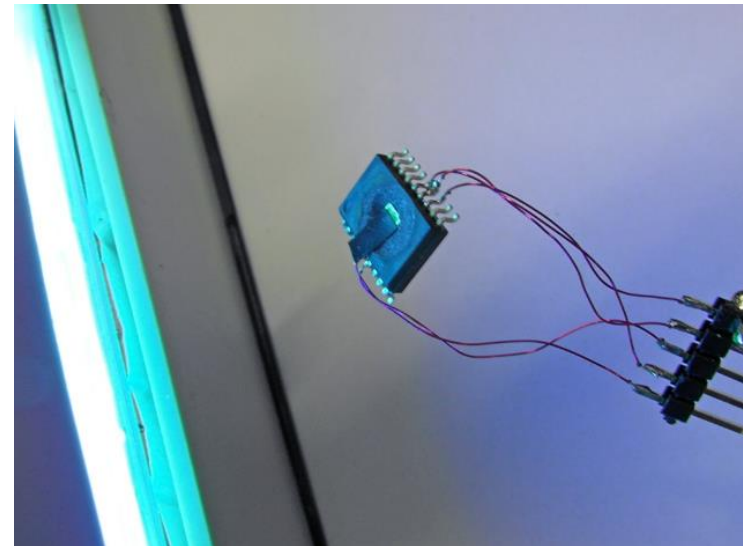
clock



laser



UV-C



A fault attack on PIN checks

```
if(check_pin(pin_entered) == true)
{
    // Do something
}
else
{
    // Error handler
}
```



```
call check_pin
cmp result, 1
bne error
// Do something
error:
// Error handler
```

The CRT-RSA algorithm

Algorithm 6 CRT-RSA signature computation

The following quantities are pre-computed once:

$$d_p \leftarrow d \bmod p - 1$$

$$d_q \leftarrow d \bmod q - 1$$

$$c_p \leftarrow q^{-1} \bmod p$$

$$c_q \leftarrow p^{-1} \bmod q$$

Then, compute:

$$x_p \leftarrow x \bmod p$$

$$x_q \leftarrow x \bmod q$$

$$s_p \leftarrow x_p^{d_p} \bmod p$$

$$s_q \leftarrow x_q^{d_q} \bmod q$$

Recombine result:

$$s \leftarrow [q \cdot c_p] \cdot s_p + [p \cdot c_q] \cdot s_q \bmod n$$

The Bellcore attack

- Assumptions:
 - CRT-RSA used
 - Fault in either one of the sub-exponentiations
 - We have valid and invalid signature on same x
- Then: n can be factored (RSA **broken**) as:
$$q = \gcd(s - \bar{s}, n), p = \frac{n}{q}$$
- There is also a variant that works with the faulty signature only (“Lenstra attack”)

Bellcore attack

$$q = \gcd(s - \bar{s}, n), p = \frac{n}{q}$$

Example: public values $n = 143$, $e = 7$, $x = 15$

Internal: $c_p = 6$, $c_q = 6$, $d = 103$, $p = 11$, $q = 13$

Valid signature $y = x^d \bmod n = 141$

Internal computation:

$$x_p = x \bmod p = 15 \bmod 11 = 4$$

$$d_p = d \bmod (p-1) = 103 \bmod 10 = 3$$

$$s_p = x_p^{d_p} \bmod p = 4^3 \bmod 11 = 9$$

(same for $s_q = 11$)

$$s = 13 * 6 * 9 + 11 * 6 * 11 = 141 \bmod 143$$

Bellcore attack

$$q = \gcd(s - \bar{s}, n), p = \frac{n}{q}$$

Faulty signature $\bar{s} = 115$

Internal computation:

$$\bar{s} = 13 * 6 * 5 + 11 * 6 * 11 = 115 \bmod 143$$

Recover q :

$$\begin{aligned} q &= \gcd(141 - 115, 143) = \gcd(26, 143) \\ &= \gcd(26, 143 - 5 * 26) = \gcd(26, 13) = 13 \end{aligned}$$

Demo: breaking CRT-RSA on Intel SGX

```
bagger> dog Enclave/encl
```

Intel SGX: Overview



- HW + ucode extensions to run **arbitrary** x86_64 code in “enclaves”
- **Goal:** Protect against adversary with root and even physical access (memory encryption)

GOOGLE CLOUD PLATFORM

Introducing Asylo: an open-source framework for confidential computing

[View on GitHub](#)

Open Enclave SDK

Build Trusted Execution Environment based applications to help protect data in use with an open source SDK that provides consistent API surface across enclave technologies as well as all platforms from cloud to edge.

[Versions](#)

Microsoft Azure

[Overview](#) [Solutions](#) [Products](#) [Documentation](#) [Pricing](#) [Training](#) [Marketplace](#) [Partners](#) [Support](#) [Blog](#) [More](#)

Azure confidential computing

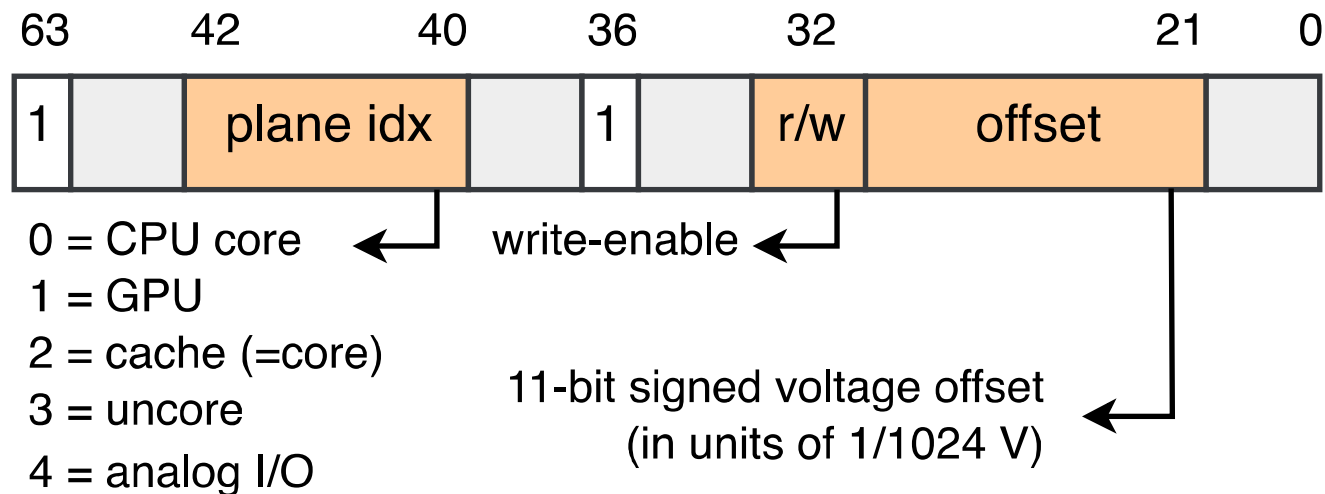
Protect and secure your cloud data while it's in use

[Start free >](#)



Undervolting Intel CPUs?

- Software-exposed interface in `msr 0x150`
- Can control relative undervolting or set an absolute voltage



A simple proof-of-concept

```
uint64_t multiplier = 0x1122334455667788;  
uint64_t correct    = 0xdeadbeef * multiplier;  
uint64_t var        = 0xdeadbeef * multiplier;
```

```
// start undervolting  
while ( var == correct )  
{  
    var = 0xdeadbeef * multiplier;  
}
```

```
// stop undervolting
```

```
// Can we ever get here? No! Yes!!
```

```
uint64_t flipped_bits = var ^ correct;
```

Faulting multiplications on Intel CPUs

Some countermeasures

- Detection-based:
 - Detect injection of fault (monitor environmental conditions)
 - Detect faulty result: compute twice (or more) and compare. Either in parallel or sequential.
- Algorithmic:
 - RSA / ECC: Verify signature after signing
 - Randomization in time (harder to inject fault)
 - “Infective” computation: fault randomizes result

Thanks!

Questions / discussion?

d.f.oswald@cs.bham.ac.uk