Security: Lecture 3

Attacking all the things!

Clémentine Maurice
L3 ENS - 2020/2021

Projects

Don’t forget to send me the ordered list of your preferences **on October 5**!

https://cmaurice.fr/teaching/ENS/

Today's lecture

- Attacking software: intro to buffer overflows
- Attacking hardware: intro to side-channel attacks

Attacking software

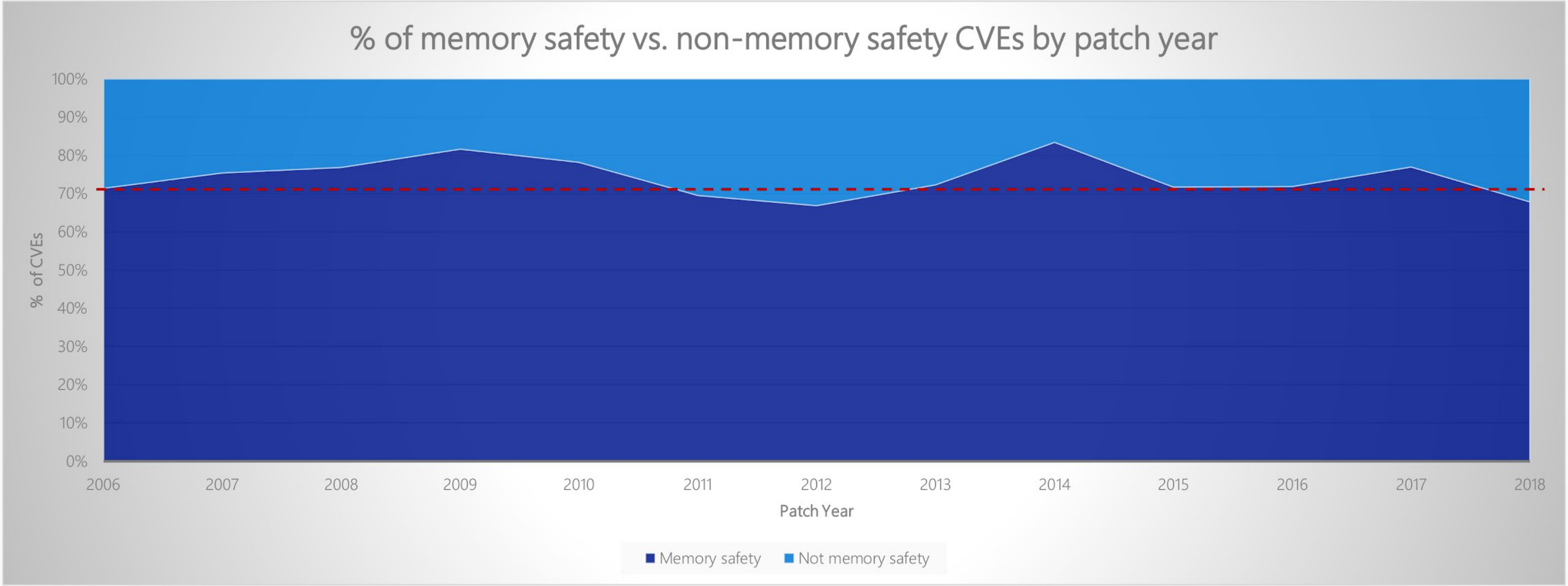
Memory corruption and buffer overflows

Memory corruption: does it really matter?

- Attacks known since ~30 years, heavily exploited since 20 years
- Why isn’t the problem solved?
- We know some **solutions**
  - design software with a safe language, check bounds
  - compiler techniques
  - system-level techniques
- None
  - solve **all problems**
  - are **practical** enough
  - are **deployed everywhere**

Memory safety issues remain dominant

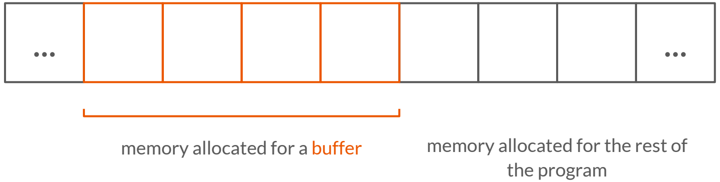
We closely study the root cause trends of vulnerabilities & search for patterns



~70% of the vulnerabilities addressed through a security update each year continue to be memory safety issues

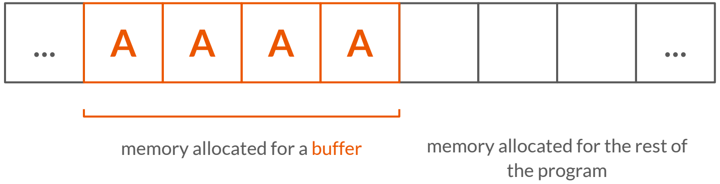
Source: Matt Miller, Microsoft (2019)

Buffer overflow: concept



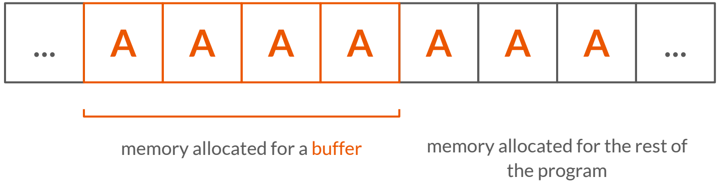
- program allocates memory for a buffer

Buffer overflow: concept



- program allocates memory for a buffer
- program writes in the buffer

Buffer overflow: concept

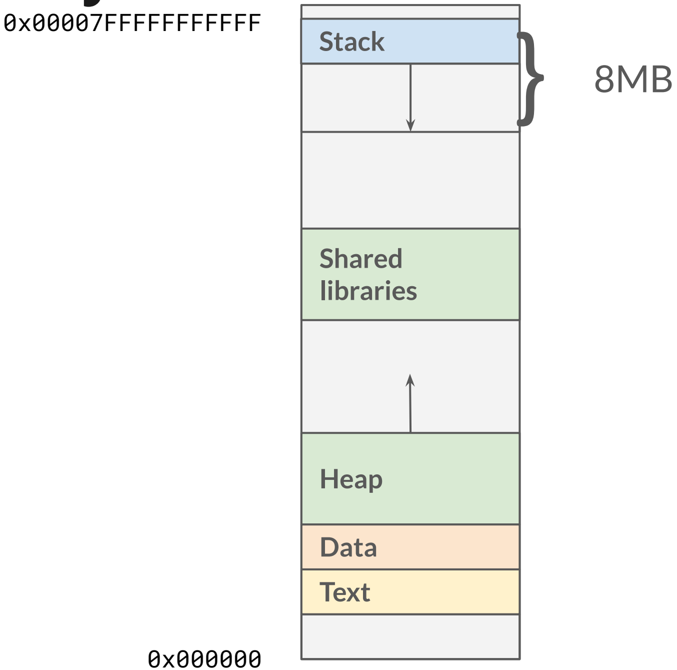


- program allocates memory for a buffer
- program writes in the buffer
- and overflows that buffer, **overwriting other parts of the program**

That's seriously the whole concept

Anatomy of a program in memory

- Stack
  - call stack (8MB limit)
  - arguments, **return address**, local variables of each function
- Heap
  - dynamically allocated as needed
- Data
  - statically allocated data (global vars, static vars, constants)
- Text/shared libraries
  - executable machine instructions, read-only



The stack

- Program = sequences of instructions to execute
- Logically divided in **functions** that call each other
- Which instruction to execute?
  - usually the next address
  - not the case if there is a function call → **when a function returns, CPU needs to know where to go back to**
- The call stack keeps track of that!
- Call stack = LIFO (last in, first out), composed of different stack frames (for each function)
- **Stack frame** = arguments, return address, local variables

Stack overflow, buffer overflow...

Stack overflow ≠ buffer overflow ≠ stack-based buffer overflow

- **Stack overflow**: execution stack grows beyond the memory that is reserved for it (e.g. recursion that never ends)
- **Buffer overflow**: a program writes beyond the end of the memory allocated for any buffer
  - **stack-based buffer overflow** → buffer is based on the stack (“classic” buffer overflow, example to come)
  - **heap-based buffer overflow** → buffer is based on the heap (more complicated)

Sample program: code

```
#include <string.h>
#include <stdlib.h>
#include <stdio.h>

void foo (char* request_from_user) {
  volatile int admin;
  char buffer[4];
  admin = 0;
  strcpy(buffer, request_from_user);
  if(admin != 0){
    printf("you are super admin\n");
  } else {
    printf("try again!\n");
  }
}
```

```
int main (int argc, char **argv) {
  foo(argv[1]);
  exit(EXIT_SUCCESS);
}
```

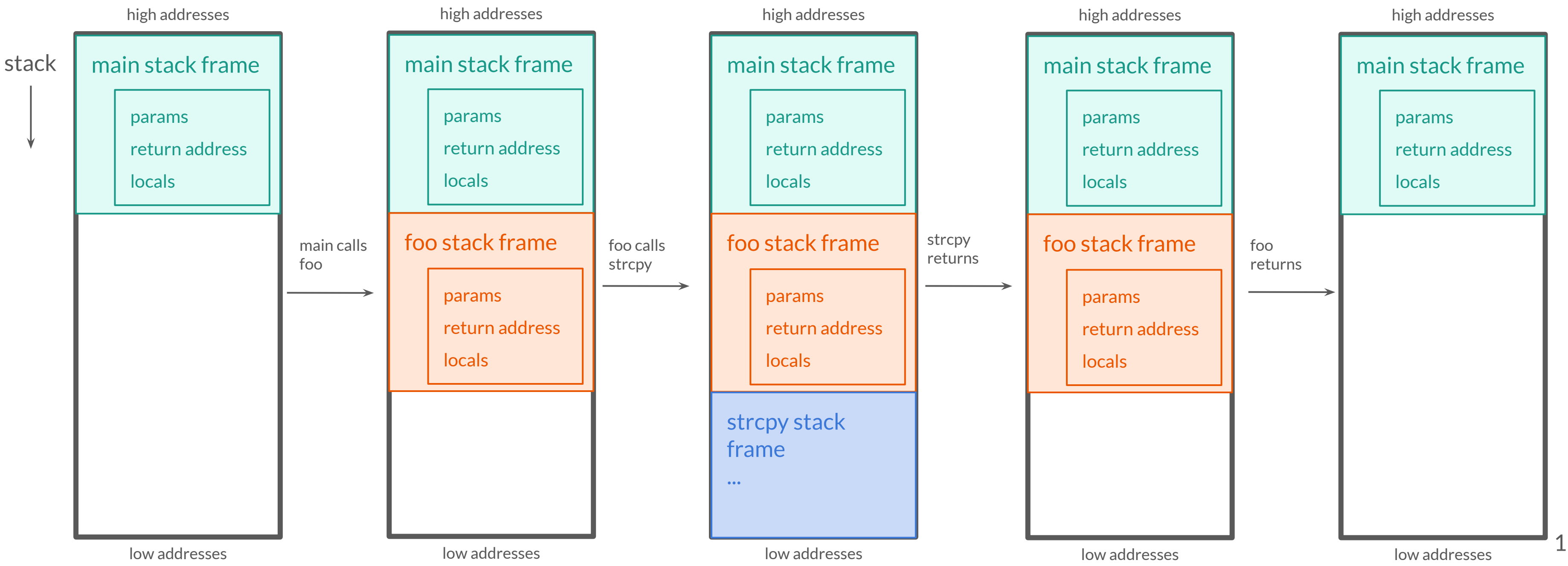
Sample program: execution

Compile with `gcc -fno-stack-protector -g -o test1 test1.c`

Run:

```
$ ./test1 1234
try again!
```

What happens in memory?

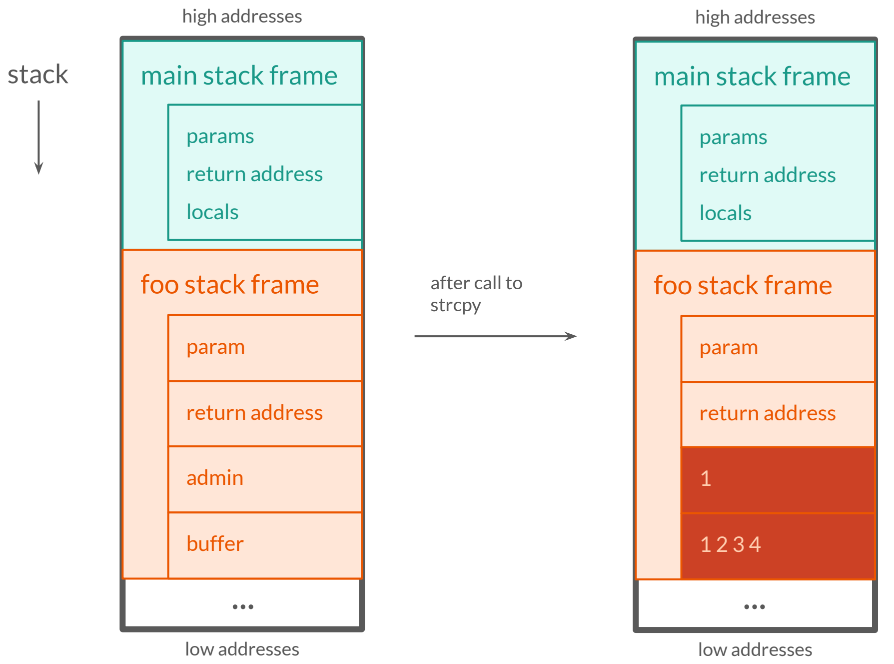


Stack-based buffer overflow 101

```
$./test1 12341
you are now super admin
```

What happened?

Stack-based buffer overflow 101



Writing out of the bounds of buffer **corrupted the “admin” variable** on the stack

Can we do better?

So far we have corrupted one variable, any other ideas of what we can do?

Can we do better?

So far we have corrupted one variable, any other ideas of what we can do?

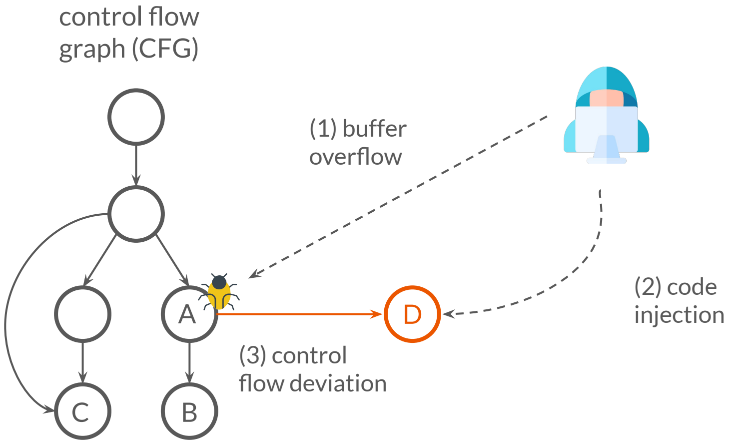
Let’s corrupt the return address!

Two main attack techniques

code **injection** attacks

code **reuse** attacks

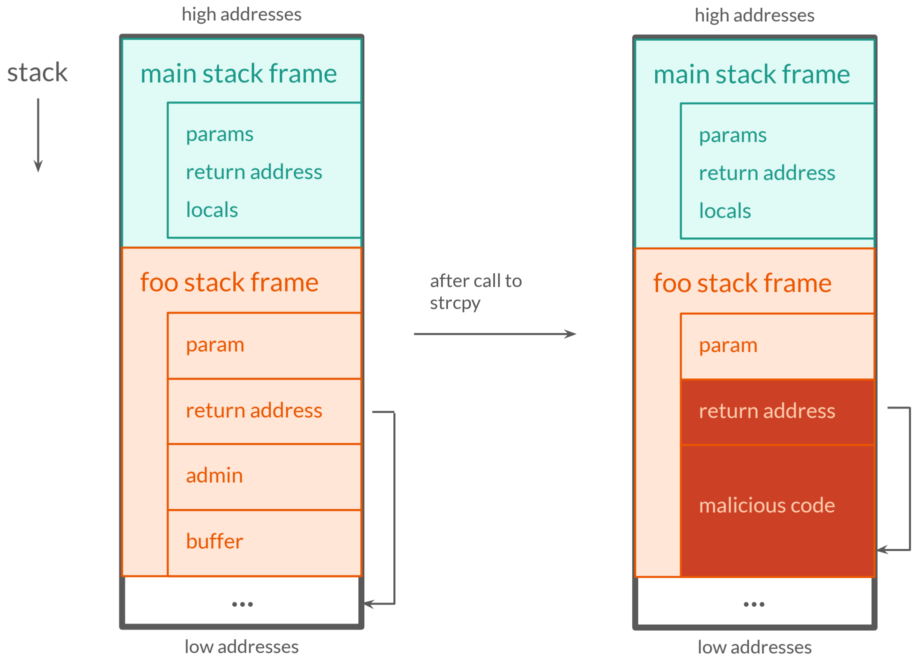
Code injection attacks: general principle



Code injection = adding a **new node** to the CFG

- Adversary can execute arbitrary malicious code
    - open a remote console (classical shellcode)
    - exploit further vulnerabilities in the OS kernel to install a virus or a backdoor

Code injection attacks



stack

high addresses

main stack frame

params
return address
locals

foo stack frame

param

return address

admin

buffer

...

low addresses

after call to strcpy

high addresses

main stack frame

params
return address
locals

foo stack frame

param

return address

malicious code

...

low addresses

- Writing out of the bounds of buffer **corrupted the return address**
- The attacker injects malicious code inside the buffer
- The return address now points to the malicious code

What does the attacker executes?

What does the attacker executes?

You have one wish, what do you do?

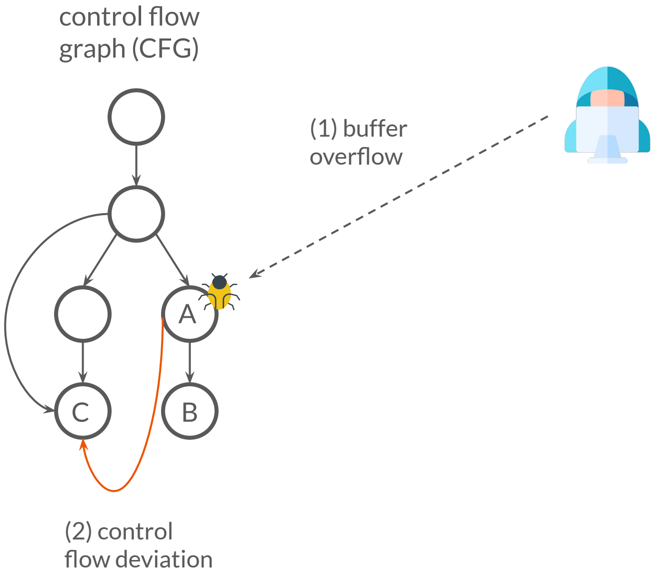
What does the attacker executes?

You have one wish, what do you do?

Wish for more wishes!

The attacker usually launches a **shell**, i.e., a program that interfaces between the user and the OS services
→ the attacker can now launch any program

Code reuse attacks: general principle



Code reuse = adding a **new path** to the CFG

- Adversary is limited to the code nodes that are available in the CFG
- Typically, the adversary will chain pieces of code (gadgets) together to execute **arbitrary code**

Code injection is more powerful

So why using code reuse attacks?

Mitigations

- Stack canaries
    - **insert a known random value** (the “canary”) on the stack before the return address
    - compiler inserts code that adds the canary and checks the canary value before using the return address
    - → canaries can be guessed, obtained with memory leaks
- Address-Space Layout Randomization
    - **randomize start or base address** of program code, libraries code, heap/stack/data regions
    - → memory leaks used to learn memory layout
- Non executable memories (NX/DEP)
    - memory is **either writable or executable** but not both (W xor X)
    - → defeated by return-to-libc attacks and Return Oriented Programming (ROP) = **code reuse attacks**

In practice, supporting NX + ASLR + canaries makes attacks much harder but **isn’t bullet proof**!

Sample program: let's try again

Compile with `gcc -g -o test1 test1.c`

(We remove the `-fno-stack-protector` from last time)

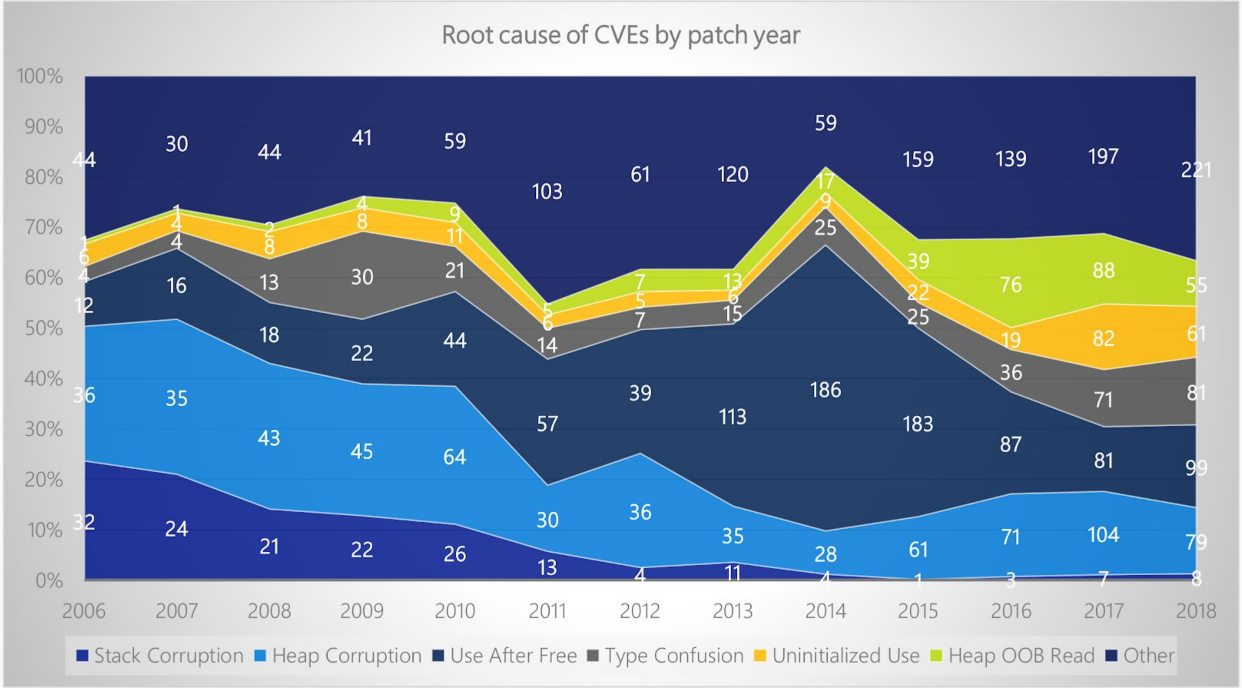
Run:

```
$ ./test1 12341
try again!
*** stack smashing detected ***: terminated
[1]  123865 abort (core dumped)  ./test1 12341
```

Other mitigations?

- How about stop using languages that allow such things? **Memory-safe** computer languages
    - Python
    - Java
    - C#
    - JavaScript
    - Go
    - Rust
- Note: doesn't mean the programs are safe, you can write insecure programs in any language
- So why do we continue using C/C++?
    - performance
    - legacy code
    - performance

Drilling down into root causes



Stack corruptions are essentially dead

Use after free spiked in 2013-2015 due to web browser UAF, but was mitigated by Mem GC

Heap out-of-bounds read, type confusion, & uninitialized use have generally increased

Spatial safety remains the most common vulnerability category (heap out-of-bounds read/write)

Top root causes since 2016:

- #1: heap out-of-bounds
- #2: use after free
- #3: type confusion
- #4: uninitialized use

Source: Matt Miller, Microsoft (2019)

Note: CVEs may have multiple root causes, so they can be counted in multiple categories

More memory corruption

- Integer overflow
- Use after free
- Heap-based buffer overflow
- ...

All the fun is in project #7!

Attacking hardware

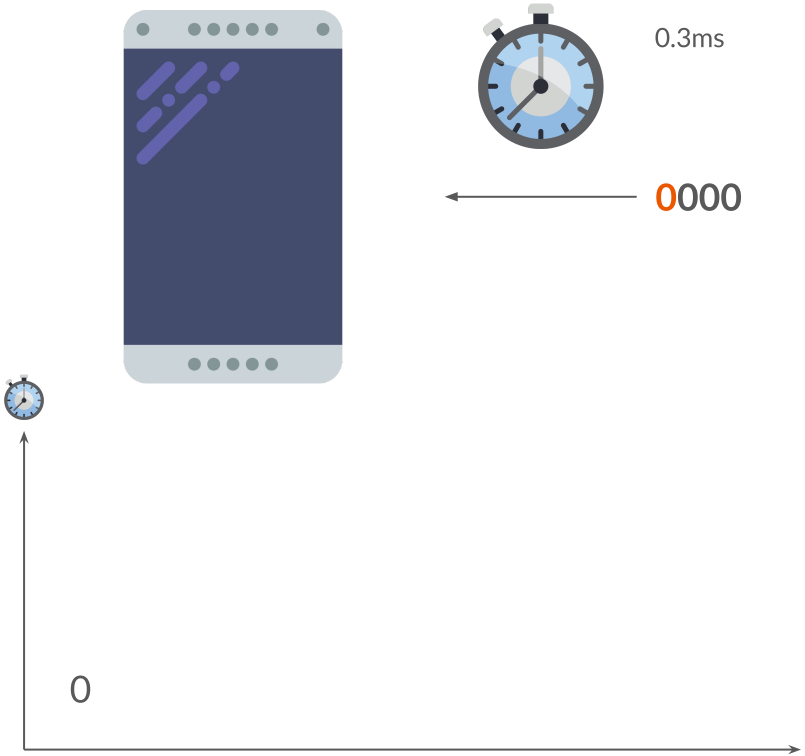
Side-channel attacks

PIN code verification

```
bool testPIN(int code [4]) {
  for (int i=0; i<4; i++) {
    if (code[i] != code_ref[i])
      return false;
    }
  return true;
}
```

PIN code verification

```
bool testPIN(int code [4]) {
  for (int i=0; i<4; i++) {
    if (code[i] != code_ref[i])
      return false;
    }
  return true;
}
```



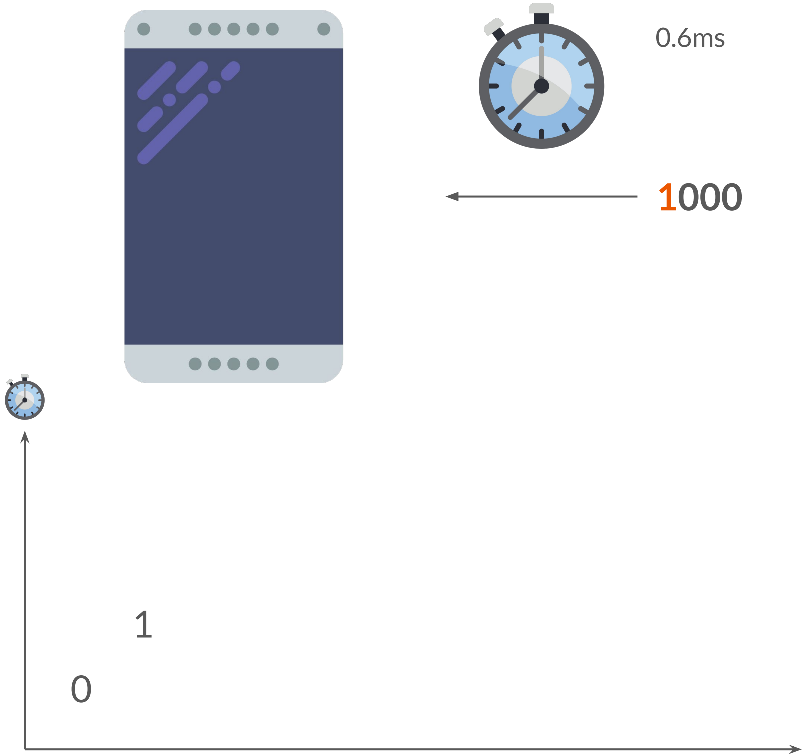
0.3ms

0000

0

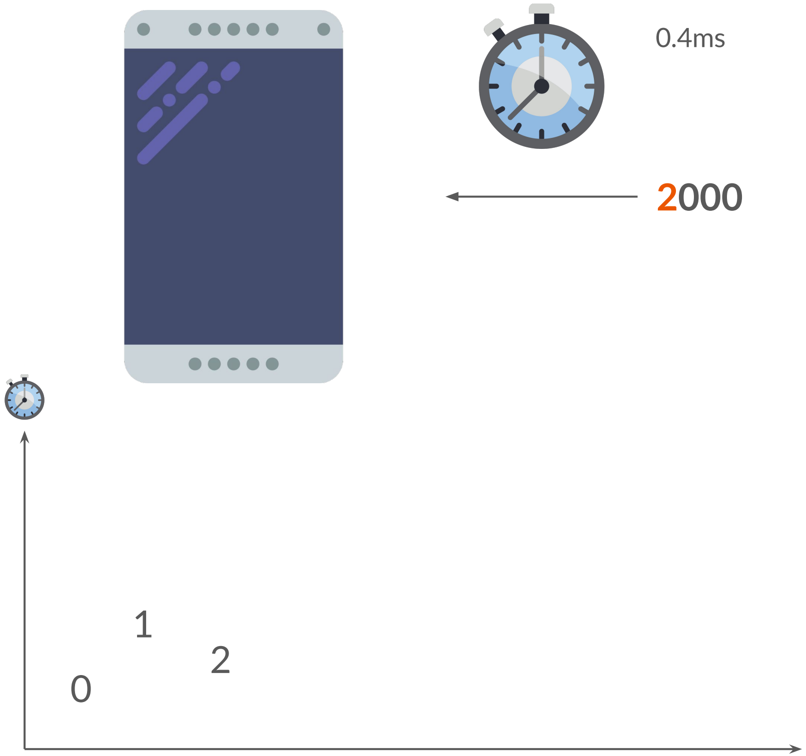
PIN code verification

```
bool testPIN(int code [4]) {
  for (int i=0; i<4; i++) {
    if (code[i] != code_ref[i])
      return false;
    }
  return true;
}
```



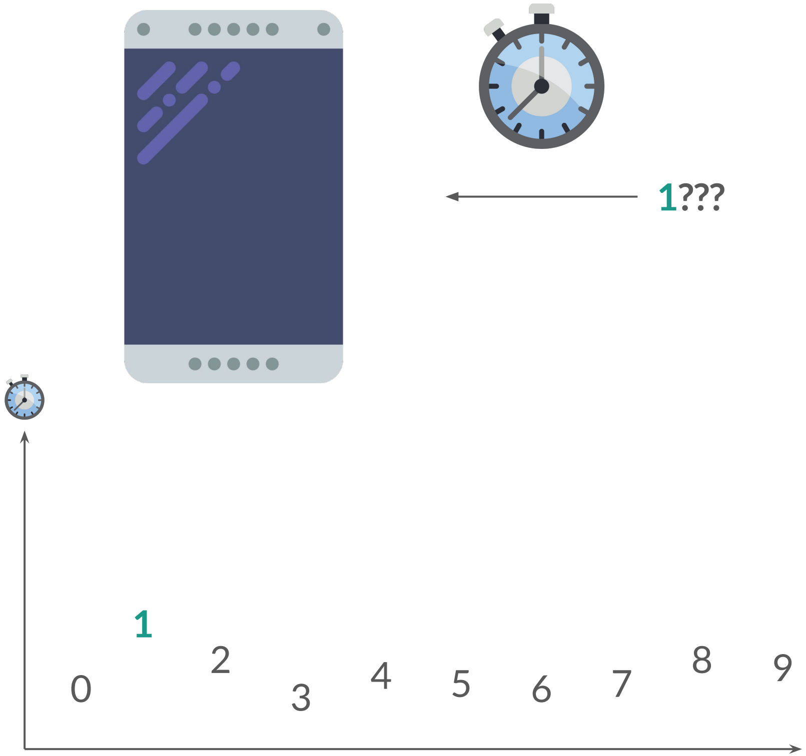
PIN code verification

```
bool testPIN(int code [4]) {
  for (int i=0; i<4; i++) {
    if (code[i] != code_ref[i])
      return false;
    }
  return true;
}
```



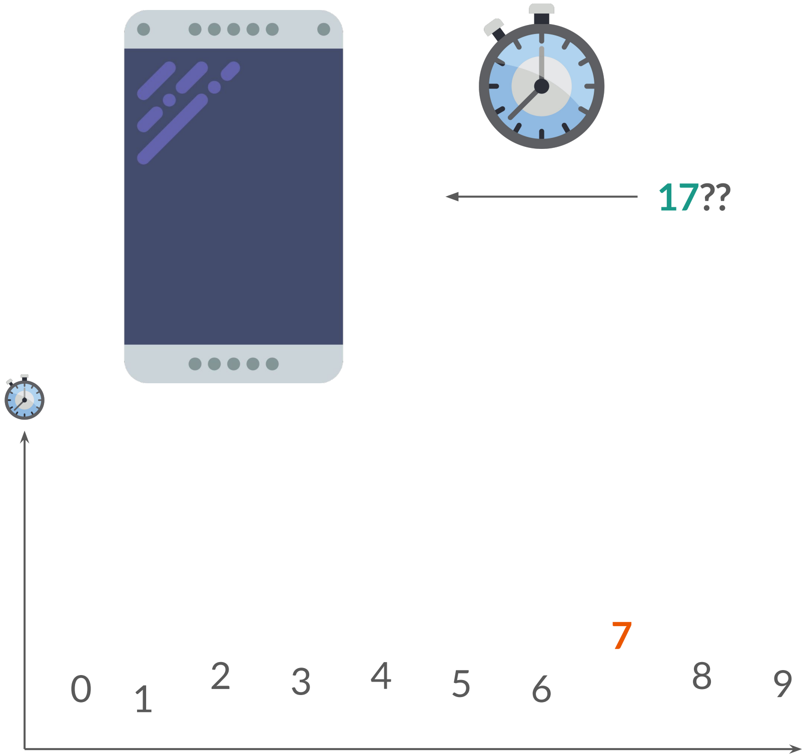
PIN code verification

```
bool testPIN(int code [4]) {
  for (int i=0; i<4; i++) {
    if (code[i] != code_ref[i])
      return false;
    }
  return true;
}
```



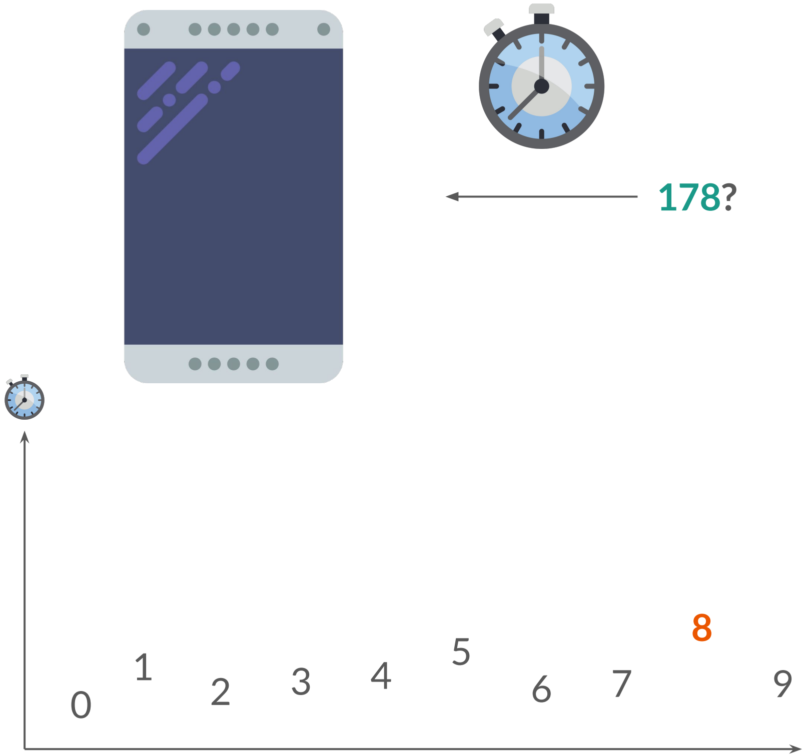
PIN code verification

```
bool testPIN(int code [4]) {
  for (int i=0; i<4; i++) {
    if (code[i] != code_ref[i])
      return false;
    }
  return true;
}
```



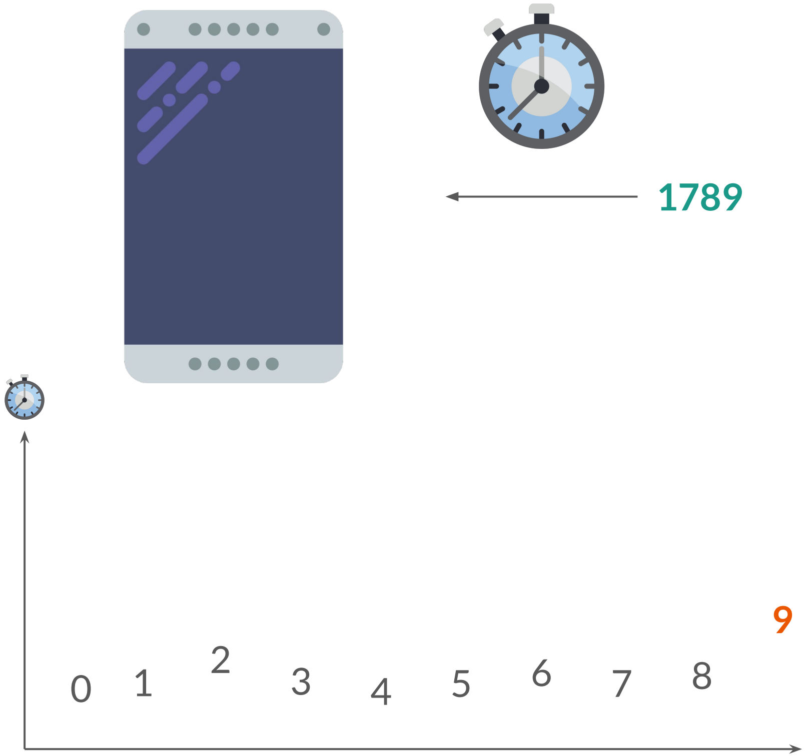
PIN code verification

```
bool testPIN(int code [4]) {
  for (int i=0; i<4; i++) {
    if (code[i] != code_ref[i])
      return false;
    }
  return true;
}
```



PIN code verification

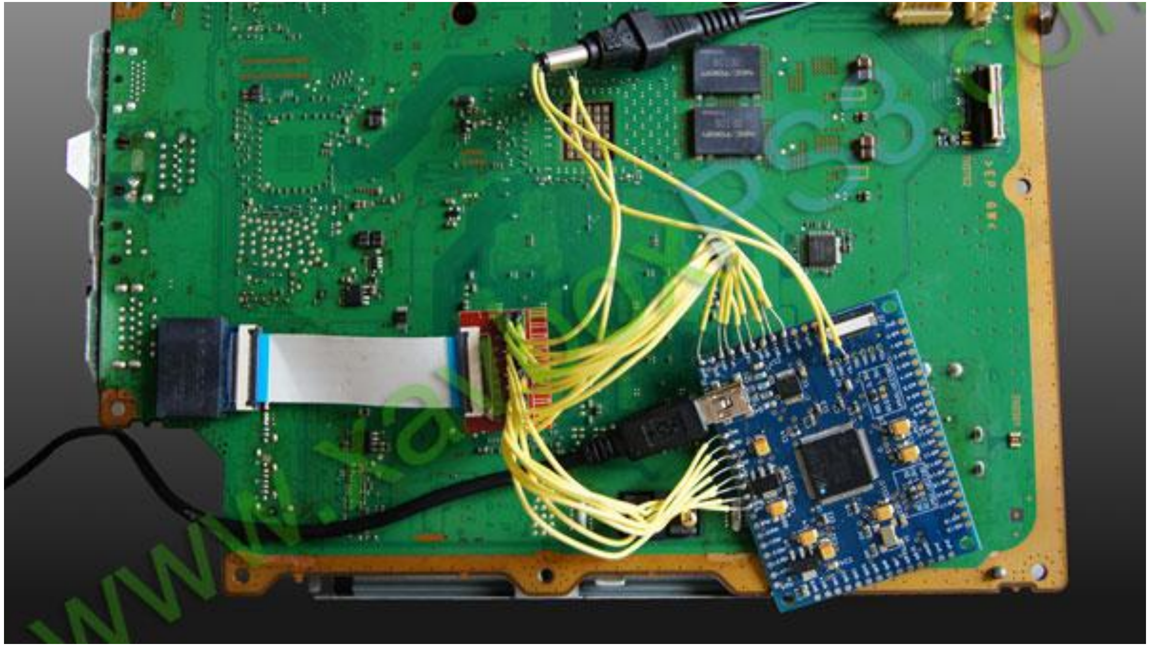
```
bool testPIN(int code [4]) {
  for (int i=0; i<4; i++) {
    if (code[i] != code_ref[i])
      return false;
    }
  return true;
}
```



In real life...

- **Computations are not running in a vacuum**, they are running on actual hardware
- If an attacker has access to the device, or can run programs on the device, **threat model** changes
- Attacker can
    - **interfere** with hardware
    - observe **side effects**

Example: PS3 firmware modification



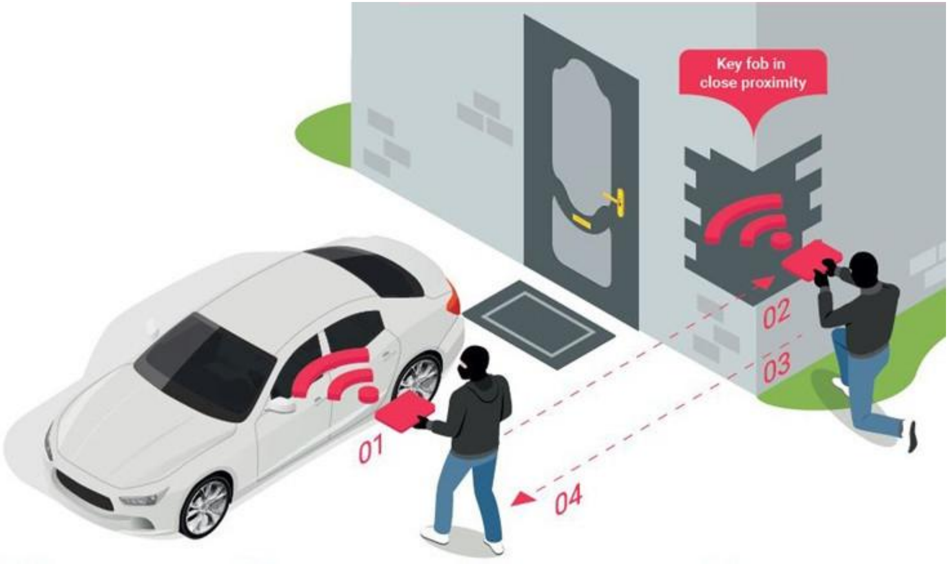
Example: PS3 firmware modification

What happened?

1. **Dump code**: requires physical access or a software vulnerability
2. Understand what it does: **reverse-engineering** tools
3. Modify it to **remove the security**
4. **Reload code**

More console hacking: https://media.ccc.de/search/?q=console+hacking

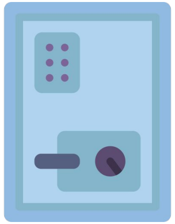
Example: Relay attacks



- One thief near the car, the other near the key
- **Capture the signal** of the key and **relay it to the car**, as if the key was close to the car
- Use of radio amplification to boost the signal of the key

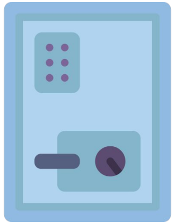
The security of a product cannot rely only on software tests

Physical attacks



How to attack a vault?

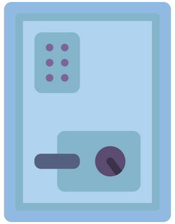
Physical attacks



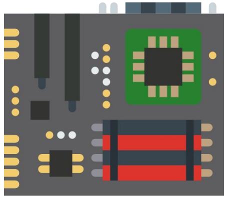
active attacks: destroying the vault

passive attacks: listening to the vault internal mechanisms

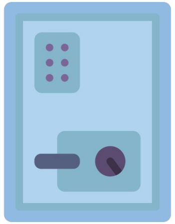
Physical attacks



- **active** attacks: destroying the vault
- **passive** attacks: listening to the vault internal mechanisms

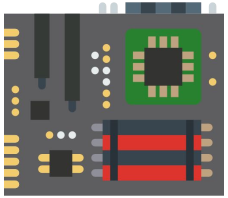


Physical attacks



active attacks: destroying the vault

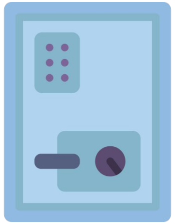
passive attacks: listening to the vault internal mechanisms



active attacks: laser, varying temperature, clock glitching, hardware trojans...

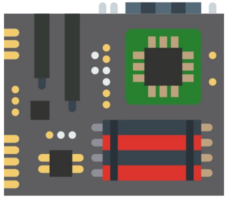
passive attacks: timing, power consumption, electromagnetic radiation...

Physical attacks



active attacks: destroying the vault

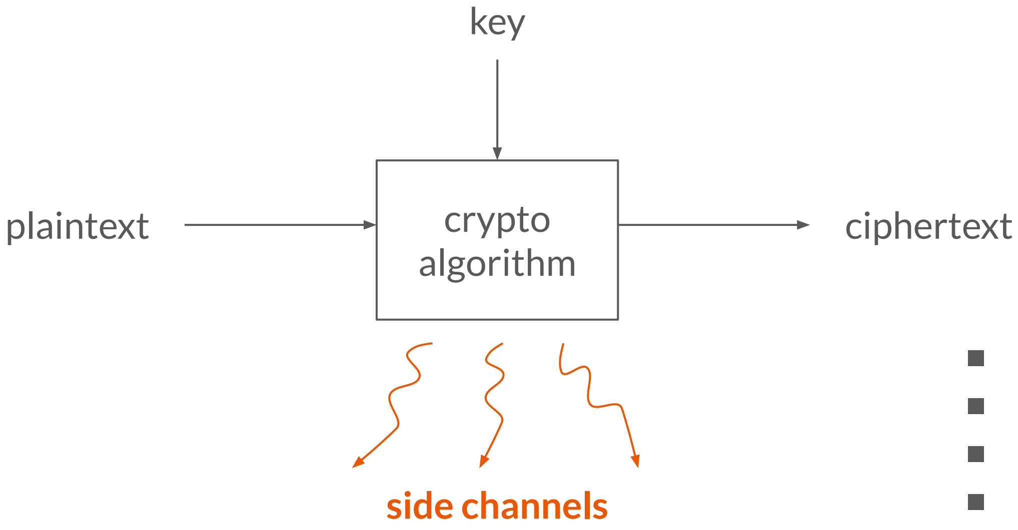
passive attacks: listening to the vault internal mechanisms



active attacks: laser, varying temperature, clock glitching, hardware trojans...

passive attacks: timing, power consumption, electromagnetic radiation...

Side-channel attacks in a nutshell



key

plaintext

crypto algorithm

ciphertext

side channels

- Timing
- Power consumption
- Electromagnetic radiation (EM)
- Sound
- ...

Side channels

- Exploits the **implementation** of a system
- Based on channels that are **outside of the software functional specification**, i.e., that are not supposed to carry useful information
- However these channels can **leak secret information**
- Usually based on some “physical” channel, e.g., timing, power consumption, EM, sound, light...

Hardware-based and software-based

Physical access

- on embedded devices

Remote access/co-located software

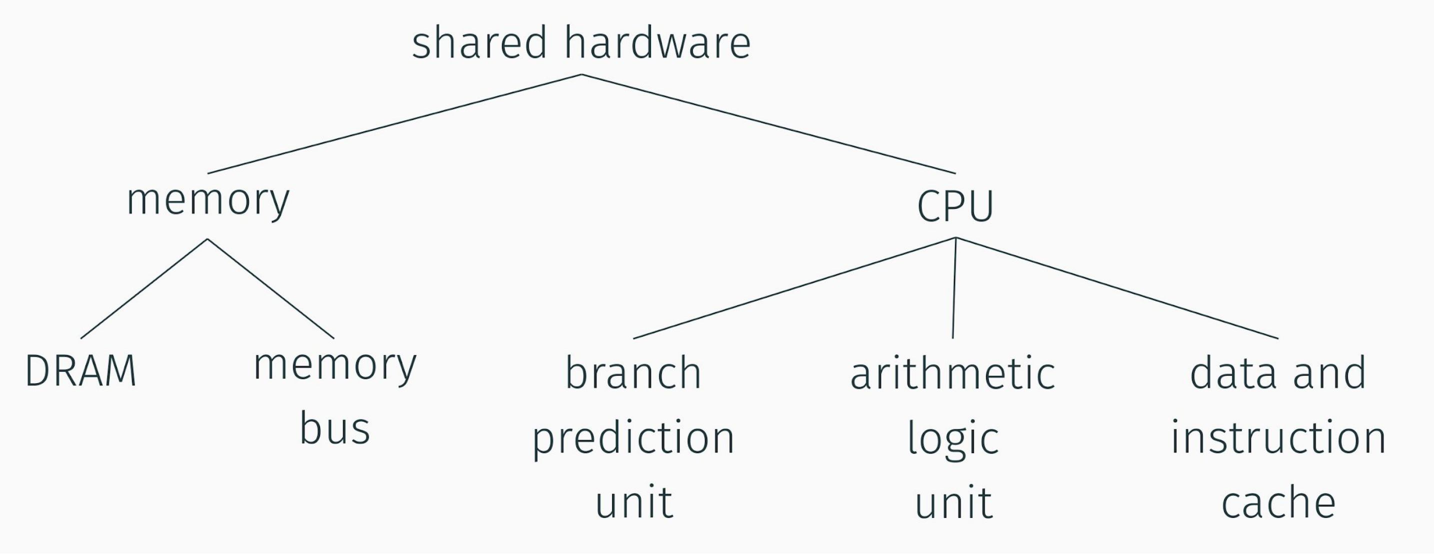
- on more complex machines

Software-based side channels

2008	Nehalem
2012	Sandy Bridge
2013	Ivy Bridge
2014	Haswell
2015	Broadwell
2016	Skylake
2017	Kaby Lake
2018	Coffee Lake

- new microarchitectures yearly
- performance improvement ≈ 5%
- very **small optimizations**: caches, branch prediction...
- ... leading to side channels
- **no documentation** on this intellectual property

Software-based side channels



shared hardware

memory

CPU

DRAM

memory bus

branch prediction unit

arithmetic logic unit

data and instruction cache

Cache attacks

- Exploit timing differences of memory accesses
  - data is in the **cache** → cache hit → **fast access**
  - data is **not in the cache** → cache miss → retrieve data from DRAM → **slow access**
- Attacker monitor which cache lines are accessed, **not the content**

Attacking an RSA implementation

Generating an RSA encryption system requires the following steps:

- randomly selecting two prime numbers *p* and *q* and calculating *n* = *pq*
- choosing a public exponent *e*. GnuPG uses *e = 65537*
- calculating a **private exponent** $d \equiv e^{-1} (\bmod (p-1)(q-1))$

The private key is the triple (*p*, *q*, *d*).

exponentiation

The decrypting function is $D(c) = \boxed{c^d} \bmod n$

Attacking an RSA implementation

Generating an RSA encryption system requires the following steps:

- randomly selecting two prime numbers *p* and *q* and calculating *n* = *pq*
- choosing a public exponent *e*. GnuPG uses *e = 65537*
- calculating a **private exponent** $d \equiv e^{-1}(\mathrm{mod}(p-1)(q-1))$

The private key is the triple (*p*, *q*, *d*).

exponentiation

The decrypting function is $D(c) = \boxed{c^d} \bmod n$

But multiplying *c* by itself *d* times is too slow!

RSA square-and-multiply exponentiation (1/2)

```
Algorithm 1: Square-and-multiply exponentiation
Input: base $b$, exponent $e$, modulus $n$
Output: $b^e \mod n$
$X \leftarrow 1$
for $i \leftarrow bitlen(e)$ downto 0 do
    $X \leftarrow$ multiply($X$,$X$)
    if $e_i = 1$ then
        $X \leftarrow$ multiply($X$, $b$)
    end
end
return $X$
```

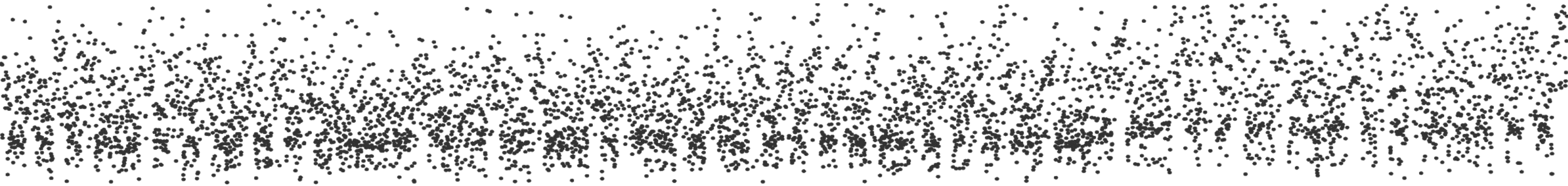
RSA square-and-multiply exponentiation (1/2)

```
Algorithm 1: Square-and-multiply exponentiation
Input: base $b$, exponent $e$, modulus $n$
Output: $b^e \mod n$
$X \leftarrow 1$
for $i \leftarrow bitlen(e)$ downto 0 do
    $X \leftarrow$ multiply($X$,$X$)
    if $e_i = 1$ then
        $X \leftarrow$ multiply($X$, $b$)
    end
end
return $X$
```

***e* is a secret value!**

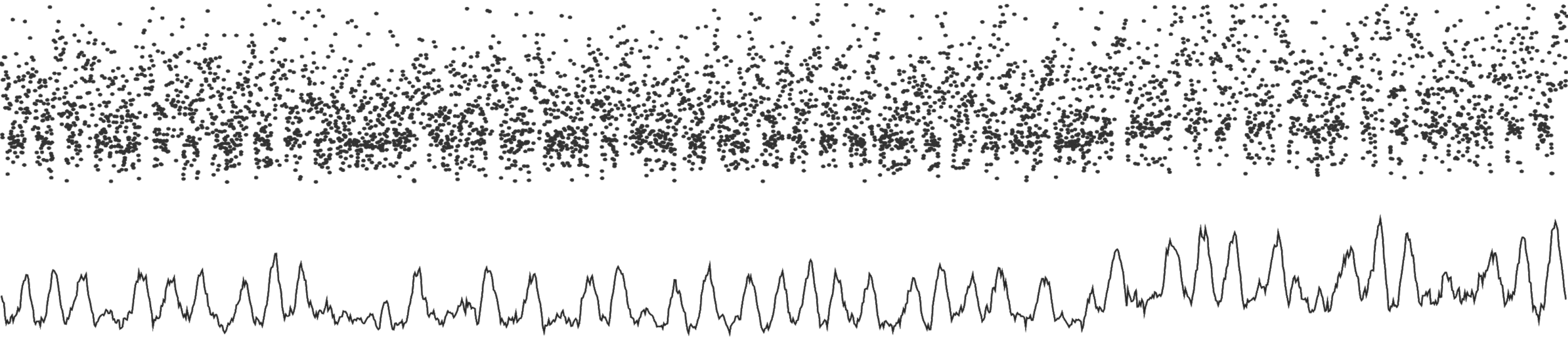
RSA square-and-multiply exponentiation (2/2)

Cache attack on the buffer holding the mutiplier *b*



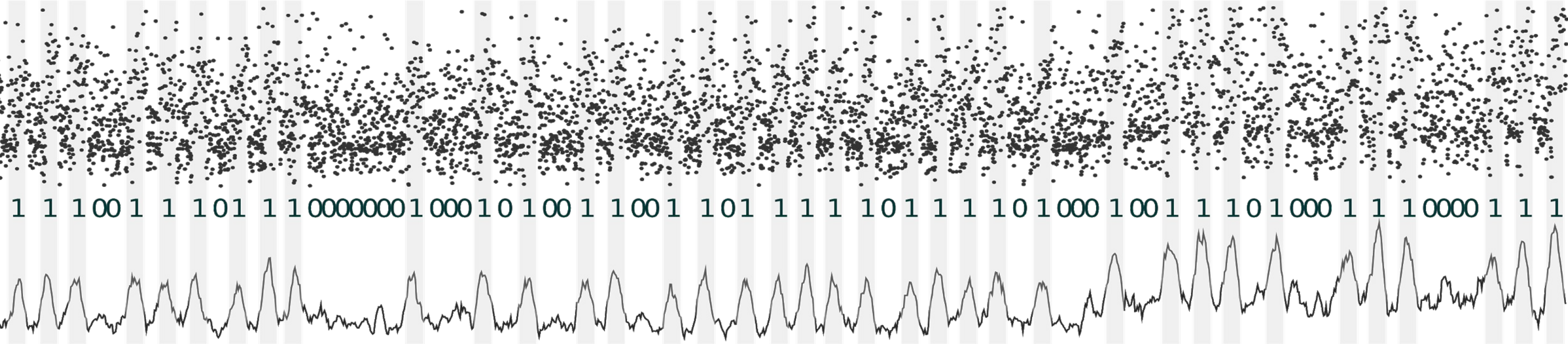
RSA square-and-multiply exponentiation (2/2)

Cache attack on the buffer holding the mutiplier *b*



RSA square-and-multiply exponentiation (2/2)

Cache attack on the buffer holding the mutiplier *b* → recovers **bits of the exponent**



Algorithm vs. Implementation

RSA, the algorithm, is not broken.

One (actually several) **implementation**(s) of RSA is (are) **broken**.

Not all implementations are created equal!

“Constant time”

- What we call “constant-time” in cryptography does not equate to constant timing
- **Constant timing is not necessary** for a secure implementation…
- … If those variations have **no relation to any secret information**
- Instead, “constant-time” means:
  - **no memory access** dependent on secret value
  - **no branches** dependent on secret value
  - no secret value as an input of instructions that are known to have a variable-time execution (e.g. DIV on x86: smaller values divide faster)

Wrapping up

Wrapping up

- Security is a **very large domain**
- Challenges ranging from the very theoretical to the very practical
- We’re not covering a lot in this lecture
- But this is why you have the **projects**!
- Presentations on **December 10**
    - from 9 to 12
    - all groups attend the presentations: mini seminar