Radically secure computing

Guillaume Bouffard [3] Sébanjila Kevin Bukasa [1] Mathieu Escouteloup [1] Alexandre Gonzalvez [1,2] Jean-Louis Lanet [1] **Ronan Lashermes** [1] Hélène Le Bouder [2] Gaël Thomas [4] Thomas Trouchkine [3]

[1]INRIA [2]IMT-Atlantique [3]ANSSI [4]DGA

18th of February, 2020

SILM Seminar, Rennes

The problem

You want to design a critical system (in a nuclear plant, an hospital, an aircraft carrier . . .): you need to chose your hardware and software.

- What processor ?
- What operating system ?

My take: you are screwed.

We need to design processors (and their ecosystem) that have much stronger security guarantees.

Content

Software

The problems of software vulnerabilities are well known

The solutions include:

- better language semantics (*e.g.* the rust language).
- Proof-oriented software (*e.g.* SeL4).
- Improved instruction set architecture (*e.g.* the CHERI project).

Software

The problems of software vulnerabilities are well known

The solutions include:

- better language semantics (*e.g.* the rust language).
- Proof-oriented software (*e.g.* SeL4).
- Improved instruction set architecture (*e.g.* the CHERI project).

But

Rice's theorem puts a limit to what we can prove after the fact. We can only prove software functionally correct if small or built with a restricted semantics.

For any generic software sufficiently large: there exists a vulnerability.

Micro-architecture

Abstract machine

Software is built relative to an abstract machine (*e.g.* the C machine), but this is not the true machine it will be executed on!

Micro-architecture

Abstract machine

Software is built relative to an abstract machine (*e.g.* the C machine), but this is not the true machine it will be executed on!

Other limits to proof-oriented software

The compiler cannot know the details of future machine implementations.
Proofs exclude common features of modern processors (*e.g.* DMA).

The gap between the true and the abstract machine creates new vulnerabilities: Spectre, Meltdown, and the dozens variants that followed.

Hardware

And then, there are the issues with the hardware itself

The attacker can alter the machine during execution: skip instructions, alter cache memories, MMU mapping, ...

Example: moving data in L2

```
0x000489b8: d65f03c0 a9be7bfd 910003fd b9001fbf
0x000489c8: b9001bbf b90017bf 900001a0 912d2000
0x000489d8: d2802002 52800001 94000b28 97fefe67
0x000489e8: d2800040 97feffe2 94008765 940087ad
0x000489f8: b9001fbf 14000010 b9001bbf 14000008
0x00048a08: 940087c1 b94017a0 11000400 b90017a0
0x00048a18: b9401ba0 11000400 b9001ba0 b9401ba0
0x00048a28: 7100c41f 54fffeed b9401fa0 11000400
```

Figure: Before fault.

```
0x000489d8: d2800040 97feffe2 00000002 00000008
0x000489e8: 00000002 00000008 910003fd b9001fbf
0x000489f8: b9001bbf b90017bf 11000400 b90017a0
0x00048a08: b9401ba0 11000400 b9001ba0 b9401ba0
0x00048a18: 7100c41f 54fffeed b9401fa0 11000400
0x00048a28: b9001fa0 b9401fa0 81040814 77777777
```

Figure: After fault.

The dilemma

High performances, low energy consumption, high security: pick one!

Most processor cores are optimized either for high performances or low energy consumption. Secure elements are coprocessors: they are dedicated to specific functionalities.

We need cores dedicated to security, with new and restricted semantics aka "radically secure computing".

Instructions

```
addi   a3, a0, 4
addi   a4, x0, 1
bltu    a4, a1, 10
jalr    x0, x1, 0
lw     a6, 0(a3)
addi   a2, a3, 0
addi   a5, a4, 0
lw     a7, -4(a2)
```

Atoms of our programs

Work on registers and offsets (encoded in the instruction value).

- Arithmetic
- Load/Store
- (Conditional) Branches
- (Unconditional) Jumps
- ...

Confidentiality

Removing side channels

- Tag some registers as `confidential`.
- If a register is `confidential`, it cannot be used as the source of a conditional branch instruction.
- If a `confidential` register is involved, all arithmetic operations have a constant and ISA-documented duration (cf ARMv8.4 DIT).
- For Load and Store instructions, the jury is still out. Forbid `confidential` registers as address sources ?
- Stricter micro-architectural state isolation guarantees, masking, ...

We propose a restricted semantics on which the compiler can base its security guarantees.

Provably hiding the control flow

The possibility to jump arbitrarily in memory is detrimental to security. If, for any reason, the attacker can control the destination, she gains the system's control.

Trapdoor predicate

Let k be a secret key and h a cryptographic hash function.

$$p(x) = \begin{cases} 1 & \text{if } h(x) = h(k) \\ 0 & \text{in the other cases} \end{cases}$$

Hiding the control flow

$$\begin{aligned} x &\leftarrow \text{user input} \\ y &\leftarrow p(x) \cdot (h(x+1) \oplus constant) \\ jump & \quad\ 0x1000 \oplus y \end{aligned}$$

Hiding properties

Properties

Without the knowledge of k, even knowing the program it is impossible to:

- find x such that we jump to an address different than $0x1000$.
- find the destination address, if we jump to somewhere different than $0x1000$.

Conclusion

The attacker has an advantage thanks to the definition of our instructions.

Forbidding indirect jumps

Why not just forbid forward indirect jumps ?

Common answers

- Some programs become impossible to write ! (Virtual method tables for OO polymorphism. . .)
- Programs become inefficient !

Forbidding indirect jumps

Why not just forbid forward indirect jumps ?

Common answers

- Some programs become impossible to write $\rightarrow$ the insecure ones.
- Programs become inefficient $\rightarrow$ introduce `dispatch` instruction to recover performance.

We can now extract a precise Control Flow Graph (CFG) from the program binary.

Issues without indirect jumps

Without indirect jumps, we cannot call a program whose address is unknown at compile time.

Some patterns become impossible

- Creating a process from the operation system.
- Calling a plug-in from an application.
- Dynamic code generation.

Issues without indirect jumps

Without indirect jumps, we cannot call a program whose address is unknown at compile time.

Some patterns become impossible

- Creating a process from the operation system.
- Calling a plug-in from an application.
- Dynamic code generation.

An indirect jump implies to switch to a new security domain.

HAPEI: Hardware-Assisted Program Execution Integrity

Program

The program is a list of instructions $\{i_0, i_1, \cdots, i_n\}$.

Encrypt at setup

k is a device-specific secret key, C is a compression function.

- Build the accumulator as the program state: $acc_0 = HMAC_k(IV)$.

$$acc_n = HMAC_k(acc_{n-1}||i_{n-1}).$$

- Encrypt the program:

$$i'_n = C(acc_n) \oplus i_n.$$

The several predecessors case

n-predecessors (cycles allowed in CFG)

The state of the program before an n-predecessor instruction must be a random invariant (rebase). We must be able to project all legitimate program states to this rebased value, and reject illegitimate values.

The several predecessors case

n-predecessors (cycles allowed in CFG)

The state of the program before an n-predecessor instruction must be a random invariant (rebase). We must be able to project all legitimate program states to this rebased value, and reject illegitimate values.

Solution: use projection into subgroups of $\mathbb{F}_{2^b}$. A subgroup of size r exists $\forall r | 2^b - 1$.
Example: $5 | 2^{16} - 1$, so there is a cyclic subgroup $\{\mu, \mu^2, \mu^3, \mu^4, \mu^5 = 1\}$ for some $\mu \in \mathbb{F}_{2^b}$ with $\mu^r = 1$.

The several predecessors case

n-predecessors (cycles allowed in CFG)

The state of the program before an n-predecessor instruction must be a random invariant (rebase). We must be able to project all legitimate program states to this rebased value, and reject illegitimate values.

Solution: use projection into subgroups of $\mathbb{F}_{2^b}$. A subgroup of size r exists $\forall r | 2^b - 1$.
Example: $5 | 2^{16} - 1$, so there is a cyclic subgroup $\{\mu, \mu^2, \mu^3, \mu^4, \mu^5 = 1\}$ for some $\mu \in \mathbb{F}_{2^b}$ with $\mu^r = 1$.

Encrypt (5-predecessors): $a_1, a_2, \ldots, a_5$. Choose random $c \in \mathbb{F}_{2^b}$. Compute polynomial P of degree 4 such that:

$$P(a_i) = c \cdot \mu^i.$$

Store $\{P, i'_n = C(c^r) \oplus i_n\}$.

Decryption

Decrypt

$$i_n = C\left(P(acc_n)^r\right) \oplus i'_n.$$

Works because $\forall i$,

$$P(a_i)^r = \left(c \cdot \mu^i\right)^r = c^r \cdot (\mu^r)^i = c^r.$$

Wrap-up

1. We need a new class of CPUs, the secure one,
2. around a new ISA with restricted semantics.
3. Indirect jumps imply to switch to a new security domain.
4. It is possible to protect against fault injection attacks with Instruction Set Randomization, but solutions are not practical today.

Thank you!

Any questions?