Understanding and fighting **fault injections** with programming languages

Sébastien MICHELLAND (UGA/LCIS, Valence)

SemSécuÉlec seminar — September 27th, 2024

Context and plan

Who are we ?

- Public research lab: Université Grenoble Alpes, in Valence.
- 3 teams (60+ researchers): computer science, automatics, electronics.

CTSYS team

Safety and security of embedded and distributed systems

High-level

OS/applications
Secure protocols

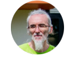
Program analysis
Compilation

Hardware security
Fault attacks

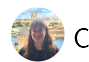
Low-level

(More people!)

Laure Gonnord


Christophe Deleuze

Sébastien Michelland

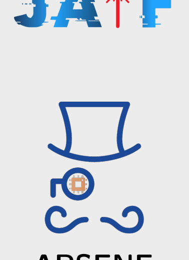

Clara Bourgeais

Vincent Beroulle

David Hély

Valentin Egloff

Maryam Esmailian



Your (second) speaker of the day

Sébastien Michelland

- Master's in theoretical C.S.; languages, compilers, formal proofs
- ... also a lot of embedded (kernel) programming
- Now 3rd-year Ph.D student at LCIS (with L. Gonnord and C. Deleuze)

- delighted to be here 😄

Outline

With a paper as running example [MDG24]:
(https://hal.science/hal-04438994)

From low-level fault modeling (of a pipeline attack) to a proven hardening scheme

Sébastien Michelland
sebastien.michelland@lcis.grenoble-inp.fr
UGA, Grenoble INP, LCIS
Valence, France

Christophe Deleuze
christophe.deleuze@grenoble-inp.fr
UGA, Grenoble INP, LCIS
Valence, France

Laure Gonnord
laure.gonnord@grenoble-inp.fr
UGA, Grenoble INP, LCIS
Valence, France

Abstract

Fault attacks present unique safety and security challenges that require dedicated countermeasures, even for bug-free programs. Models of these complex attacks are made workable by approximating their effects to a suitable level of abstraction. The common practice of targeting the Instruction Set Architecture (ISA) level isn't ideal because it discards important micro-architectural information, leading to weaker security guarantees. Conversely, including micro-architectural details makes countermeasures harder to model and reason about, creating a new challenge in validating and trusting protections.

We show that a semantic approach to modeling faults makes micro-architectural models workable, and enables precise cooperation between software and hardware in the design of countermeasures. We demonstrate the approach by designing and implementing a compiler/hardware countermeasure, which protects against a state-of-the-art pipeline fetch attack that generalizes multi-fault instruction skips. Crucially, we provide a formal security proof that guarantees faults are detected by the end of every basic block. This result shows that carefully embracing the complexity of low-level systems enables finer, more secure countermeasures

faults' effects to a desired level of abstraction. These span from bit flips in RTL (Register Transfer Level) latches [Tollec et al. 2022] to failures in pipeline forwarding [Laurent 2020] to corrupted ISA registers [Barthe et al. 2014] and branch inversion directly in source code [Potet et al. 2014]. Countermeasures are then based on these models, so in a sense secure programs resist *fault models* rather than *faults*. The clear trade-off is one of accuracy versus simplicity; low-level descriptions are more true to practical attacks, but high-level approximations make it practical (in many cases *possible*) to reason about and protect against them.

In practice, most existing works study faults at the ISA level, based on mis-executions of assembler programs (instruction skips, wrong jumps, corrupted registers, etc. [Höller et al. 2015]), with countermeasures as *transformations* of assembler programs. This is a natural choice as assembler is the lowest software abstraction, and dealing with software has benefits such as ease of deployment, board-independence, compiler automation, and the ability to protect only critical sections of programs (compared to fixed costs in e.g. die surface). Hardware protections [Lashermes et al. 2018] are less common, but better equipped to deal with local and remote side-channel attacks [Tillich et al. 2007], which share many

Context

- ▶ "Fault attacks" are super complicated
- ▶ Must approximate with "fault models"
- ▶ By comparison, software is pure math

The difficulty

- ▶ Faults break software *abstractions*
- ▶ Must work around existing technology

Elements of solutions

- ▶ Modding semantics for security
- ▶ Software/hardware collaborations

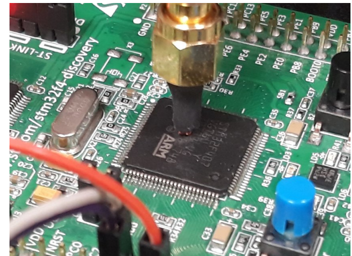
Faults and fault models

Fault injections

Fault: abnormal condition leading to incorrect behavior

$$2 + 2 \rightarrow \text{briefly short-circuit the right pins} \rightarrow 5$$

Fault injection: creating a fault on purpose



Electromagnetic fault injection [Sol+21]

Power/clock glitches, lasers, EM pulses...

⚠ **Challenges**

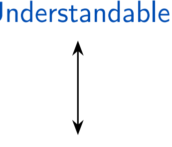
- Can hardly predict outcomes
- Some consistent behaviors
- Many very rare and very weird behaviors

Fault models

Fault model: approximate description of common fault behaviors

Examples:

- Invert an `if()` ◀ *C source*
- Corrupt program values ◀ *IR-ish*
- Skip instructions ◀ *Assembly*
- Cancel pipeline forwarding [Lau20] ◀ *Micro-arch*



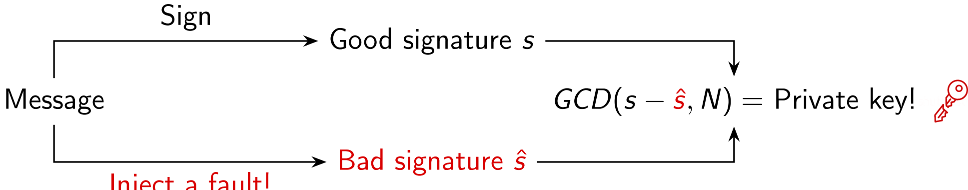
Inherent tension

Fault models are always a compromise between **accuracy** and **simplicity**.

So how bad is it for security?

VERY.

Well-known attack on RSA [Bar+06]:



General vibe: by default you assume that everything will break.

- It won't always, but the reasons why are subtle.

Rising in abstraction

Fault models in software

- Fault model: **instruction skip**

```c
int mac(int a, int b, int c) {
  return a + b * c;
}
```

Compile with LLVM -O1 for RISC-V:

```
mac:
    mul     a1, a2, a1
    add     a0, a0, a1
    ret
```

Equivalent C code if we...

Fault models in software

- Fault model: **instruction skip**

```c
int mac(int a, int b, int c) {
  return a + b * c;
}
```

Compile with LLVM -O1 for RISC-V:

```
mac:
    ~~mul     a1, a2, a1~~
    add     a0, a0, a1
    ret
```

Equivalent C code if we...

- Skip `mul`: `return a + b`

Fault models in software

- Fault model: **instruction skip**

```c
int mac(int a, int b, int c) {
  return a + b * c;
}
```

Compile with LLVM -O1 for RISC-V:

```
mac:
    mul     a1, a2, a1
    add     a0, a0, a1
    ret
```

(The `add a0, a0, a1` line is struck through.)

Equivalent C code if we...

- Skip `mul`: `return a + b`
- Skip `add`: `return a`

Fault models in software

- Fault model: **instruction skip**

```
int mac(int a, int b, int c) {
  return a + b * c;
}
```

Compile with gcc -O1 for SuperH:

```
mac:
    mul.l    r6, r5
    sts      macl, r0
    rts
    add      r4, r0
```

Equivalent C code if we...

Fault models in software

- Fault model: **instruction skip**

```c
int mac(int a, int b, int c) {
  return a + b * c;
}
```

Compile with gcc -O1 for SuperH:

```
mac:
    mul.l     r6, r5
    sts       macl, r0
    rts
    add       r4, r0
```

Equivalent C code if we...

- Skip mul.l: return a + <old_macl>

Fault models in software

- Fault model: **instruction skip**

```
int mac(int a, int b, int c) {
  return a + b * c;
}
```

Compile with gcc -O1 for SuperH:

```
mac:
    mul.l    r6, r5
    sts      macl, r0
    rts
    add      r4, r0
```

(The line `sts macl, r0` is struck through.)

Equivalent C code if we...

- Skip `mul.l`: `return a + <old_macl>`
- Skip `sts`: `return a + <old_r0>`

Fault models in software

- Fault model: **instruction skip**

```c
int mac(int a, int b, int c) {
  return a + b * c;
}
```

Compile with gcc -O1 for SuperH:

```
mac:
    mul.l    r6, r5
    sts      macl, r0
    rts
    add      r4, r0
```

Equivalent C code if we...

- Skip `mul.l`: `return a + <old_macl>`
- Skip `sts`: `return a + <old_r0>`
- Skip add: `return b * c`

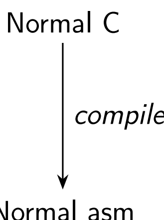
Fault models in software

- Fault model: **instruction skip**

```c
int mac(int a, int b, int c) {
  return a + b * c;
}
```

Compile with gcc -O1 for SuperH:

```
mac:
    mul.l    r6, r5
    sts      macl, r0
    rts
    add      r4, r0
```



Equivalent C code if we...

- Skip `mul.l`: `return a + <old_macl>`
- Skip `sts`: `return a + <old_r0>`
- Skip `add`: `return b * c`

Fault models in software

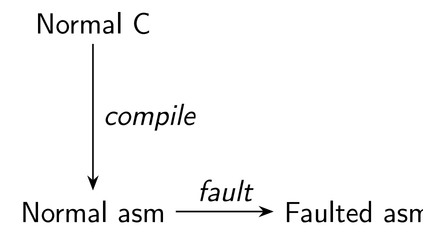
- Fault model: **instruction skip**

```c
int mac(int a, int b, int c) {
  return a + b * c;
}
```

Compile with gcc -O1 for SuperH:

```
mac:
    mul.l    r6, r5
    sts      macl, r0
    rts
    add      r4, r0
```

(The last instruction, `add r4, r0`, is struck through in red.)



Equivalent C code if we...

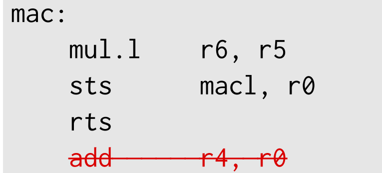
- Skip `mul.l`: `return a + ` `<old_macl>`
- Skip `sts`: `return a + ` `<old_r0>`
- Skip add: `return b * c`

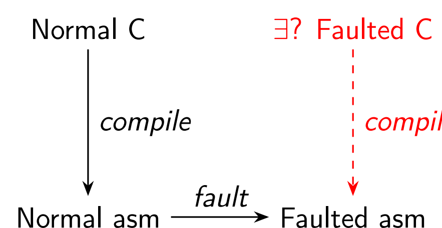
Fault models in software

- Fault model: **instruction skip**

```c
int mac(int a, int b, int c) {
  return a + b * c;
}
```

Compile with gcc -O1 for SuperH:





Equivalent C code if we...

- Skip `mul.l`: `return a + <old_macl>`
- Skip `sts`: `return a + <old_r0>`
- Skip `add`: `return b * c`

At the core of the software stack: abstractions

Why is "`return a + <old_r0>`" not a valid C program?

At the core of the software stack: abstractions

Why is "`return a + <old_r0>`" not a valid C program?

Abstraction: act of simplifying notions by hiding irrelevant details

1. Abstraction **hides** system details from you
2. Abstraction **controls** these details for you

At the core of the software stack: abstractions

Why is "`return a + <old_r0>`" not a valid C program?

Abstraction: act of simplifying notions by hiding irrelevant details

1. Abstraction **hides** system details from you
2. Abstraction **controls** these details for you

e.g. assembly *registers* become *integer variables* in C

- C allocates variables to registers/memory, you can't choose
- C variables can *(almost)* never be used uninitialized

At the core of the software stack: abstractions

Why is "`return a + <old_r0>`" not a valid C program?

Abstraction: act of simplifying notions by hiding irrelevant details

1. Abstraction **hides** system details from you
2. Abstraction **controls** these details for you

e.g. assembly *registers* become *integer variables* in C

- C allocates variables to registers/memory, you can't choose
- C variables can *(almost)* never be used uninitialized

Fundamental problem

C only describes *some* CPU behaviors and "`return a + <old_r0>`" isn't one of them.

Reversing abstraction descent is *approximate*

		Coverage (fictional)
C source code		
...		
Many compiler IRs		
...		
ISA/Assembly	Skip an instruction	100%

Reversing abstraction descent is *approximate*

		Coverage (fictional)
C source code		
...		
Many compiler IRs	Skip an IR instruction	85%
...	↑	
ISA/Assembly	Skip an instruction	100%

Reversing abstraction descent is *approximate*

		Coverage (fictional)
C source code	Skip a C statement	10% ⚠
…	↑	
Many compiler IRs	Skip an IR instruction	85%
…	↑	
ISA/Assembly	Skip an instruction	100%

Reversing abstraction descent is *approximate*

		Coverage (fictional)
C source code	Skip a C statement	10% ⚠
...	↑	
Many compiler IRs	Skip an IR instruction	85%
...	↑	
ISA/Assembly	Skip an instruction	100%

(software)

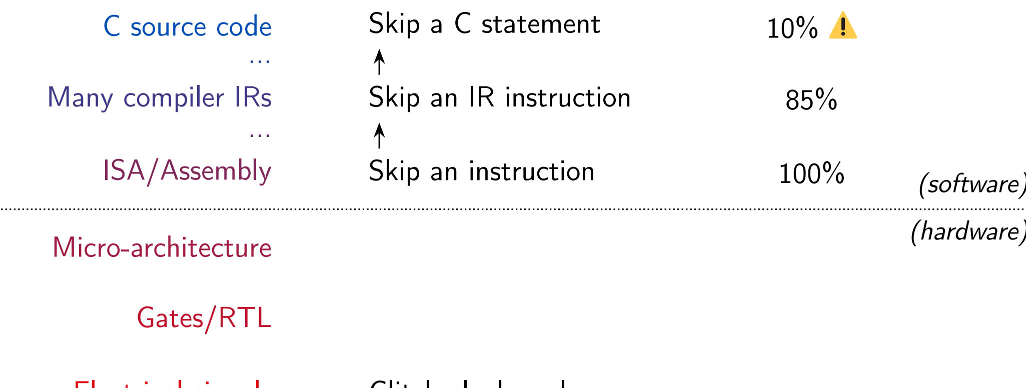
(hardware)

Micro-architecture

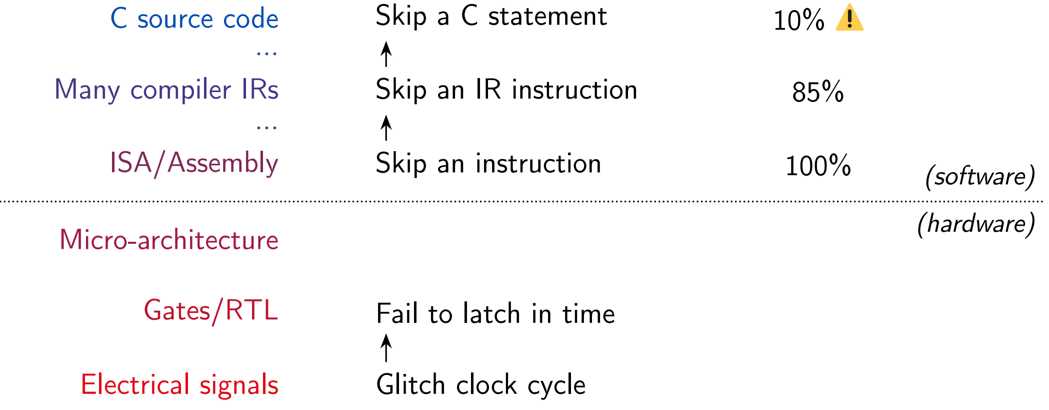
Gates/RTL

Electrical signals

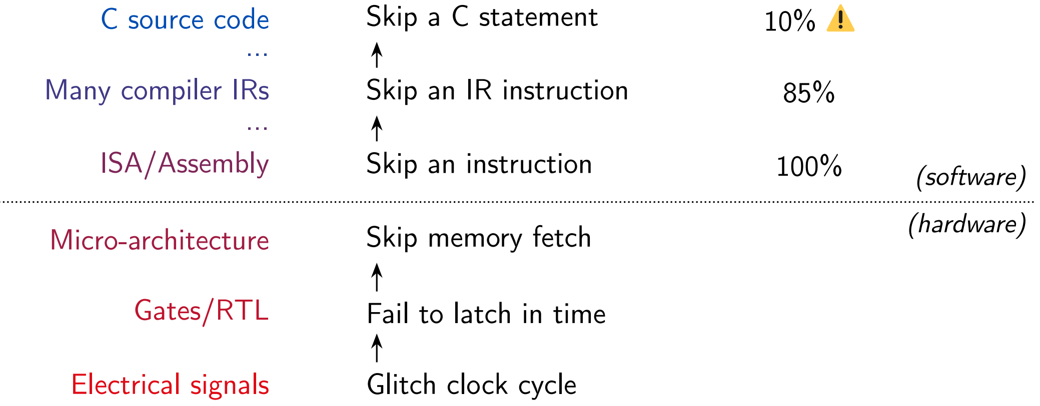
Reversing abstraction descent is *approximate*



Reversing abstraction descent is *approximate*



Reversing abstraction descent is *approximate*



Reversing abstraction descent is *approximate*

		Coverage (fictional)
C source code	Skip a C statement	3%? ⚠⚠
…	↑	
Many compiler IRs	Skip an IR instruction	25%?
…	↑	
ISA/Assembly	Skip an instruction	30%?
	↑	
Micro-architecture	Skip memory fetch	50%?
	↑	
Gates/RTL	Fail to latch in time	85%?
	↑	
Electrical signals	Glitch clock cycle	100%

Morality: the cost of rising in abstraction

Rising in abstraction is **difficult** and **approximate**.

- Hence the simplicity/accuracy compromise for fault models

Approximating undermines security guarantees:

- Software protections for models at assembly level bypassed with micro-arch abuse. [Yuc+16]

Key target

We wish for models and countermeasures that *minimize* abstraction lifts.

Language semantics to the rescue

C source code

...

Many compiler IRs

...

ISA/Assembly

(software)

(hardware)

Micro-architecture

Gates/RTL

Electrical signals

Language semantics to the rescue

C source code

...

Many compiler IRs

...

ISA/Assembly

(software)

(hardware)

Micro-architecture

Gates/RTL

Electrical signals

Behavior defined by physical objects and physical equations

Language semantics to the rescue

C source code
…
Many compiler IRs
…
ISA/Assembly

Behavior defined by program code and language rules

(software)

(hardware)

Micro-architecture

Gates/RTL

Electrical signals

Behavior defined by physical objects and physical equations

Language semantics to the rescue

C source code
...
Many compiler IRs
...
ISA/Assembly

Behavior defined by program code and language rules

Semantic rules

ALIGNED-16

$$\frac{\text{PC aligned} \qquad \text{fetch}_{32}(\text{PC}) = d \qquad \text{LSH}(d) \text{ is a 16-bit instruction}}{\langle \text{PC}, \alpha \rangle \to [\![\text{LSH}(d)]\!](\text{PC}, \alpha)}$$

(software)

(hardware)

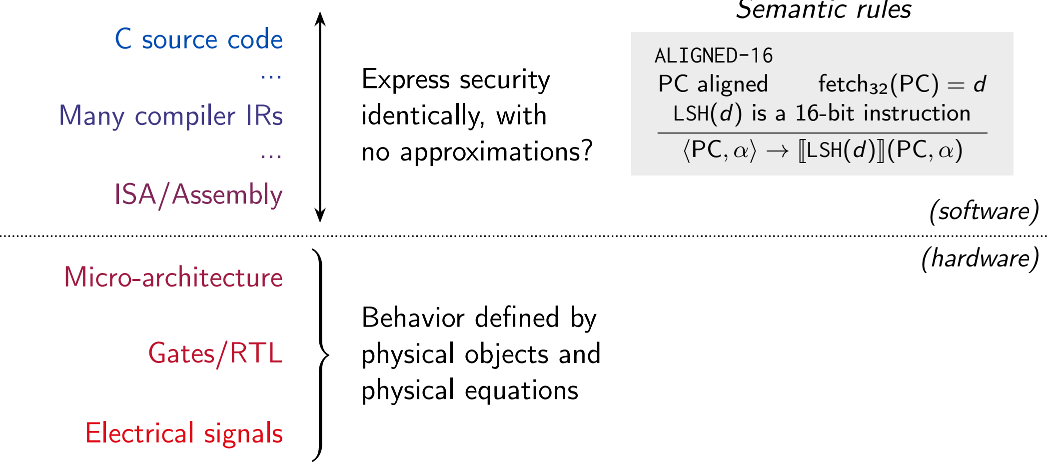
Micro-architecture

Gates/RTL

Electrical signals

Behavior defined by physical objects and physical equations

Language semantics to the rescue



In an ideal world... 🌈

We'd love to:

1. Stop fault models' approximations at assembly level or lower
2. Let software's math handle the complexity of generating secure code

In an ideal world... 🌈

We'd love to:

1. Stop fault models' approximations at assembly level or lower
2. Let software's math handle the complexity of generating secure code

In practice, still:

- Theoretical foundations for secure code generation not yet complete
- The math works on C and not much lower
- Older tech (languages/compilers) is not helping

Fetch skips

What the paper's about

- Study a **low-level fault model**
  - "Fetch skips", more accurate than instruction skips
- Design a **proven countermeasure**, aware of low-level behaviors
  - Implemented in LLVM/ld, tested in QEMU
- Based on **compiler/hardware** collaboration

Novelties?

- Countermeasure to low-level model. NEW
- Semantic model for proof. NEW
- Multi-step hardening with compiler and linker. NEW

Mechanisms of a low-level fault model: *fetch skips*.

g:	c.addi sp, sp, -16	c.sw ra, 12(sp)
g+4:	call f	(call *cont.*)
g+8:	c.addi a0, a0, 1	lw ra, 12(sp)
g+12:	(lw *cont.*)	addi sp, sp, 16
g+16:	(addi *cont.*)	c.ret

Aligned   Unaligned

```
int g(int x) {
  return f(x) + 1;
}
```

- 16-bit instructions
- Aligned 32-bit instructions
- Unaligned 32-bit instructions

Mechanisms of a low-level fault model: *fetch skips*.

g:	c.addi sp, sp, -16	c.sw ra, 12(sp)
S32	~~call f~~	~~(call *cont.*)~~

g+8:	c.addi a0, a0, 1	lw ra, 12(sp)
S&R32	c.addi a0, a0, 1	lw ra, 12(sp)
g+16:	(addi *cont.*)	c.ret

Microarchitectural-level

- **Skip 32 bits:**
  Skip a full row.
- **Skip and repeat 32 bits:**
  Replace a row with predecessor.

Found by Alshaer et al. [Als+22]

Mechanisms of a low-level fault model: *fetch skips*.

g:	`c.addi sp, sp, -16`	`c.sw ra, 12(sp)`
S32	~~`call f`~~	~~(`call` *cont.*)~~

g+8:	`c.addi a0, a0, 1`	`lw ra, 12(sp)`
g+12:	(`lw` *cont.*)	`addi sp, sp, 16`
g+16:	(`addi` *cont.*)	`c.ret`

Annoying consequences:

- **Skip one instruction**
- Skip two instructions
- Corrupt parameters
- Craft a new instruction
- Craft multiple instructions (!)

Mechanisms of a low-level fault model: *fetch skips*.

S32	~~c.addi sp, sp, -16~~	~~c.sw ra, 12(sp)~~
g+4:	call f	(call *cont.*)

g+8:	c.addi a0, a0, 1	lw ra, 12(sp)
g+12:	(lw *cont.*)	addi sp, sp, 16
g+16:	(addi *cont.*)	c.ret

Annoying consequences:

- Skip one instruction
- **Skip two instructions**
- Corrupt parameters
- Craft a new instruction
- Craft multiple instructions (!)

Mechanisms of a low-level fault model: *fetch skips*.

g:	c.addi sp, sp, -16	c.sw ra, 12(sp)
g+4:	call f	(call *cont.*)

g+8:	c.addi a0, a0, 1	lw ra, 12(sp)
S32	~~(lw *cont.*)~~	~~addi sp, sp, 16~~
g+16:	(addi *cont.*)	c.ret

→ lw ra, 16(sp)

Annoying consequences:

- Skip one instruction
- Skip two instructions
- **Corrupt parameters**
- Craft a new instruction
- Craft multiple instructions (!)

Mechanisms of a low-level fault model: *fetch skips*.

g:	`c.addi sp, sp, -16`	`c.sw ra, 12(sp)`
g+4:	`call f`	(`call` *cont.*)

S32	~~`c.addi a0, a0, 1`~~	~~`lw ra, 12(sp)`~~
g+12:	(`lw` *cont.*)	`addi sp, sp, 16`
g+16:	(`addi` *cont.*)	`c.ret`

Annoying consequences:

- Skip one instruction
- Skip two instructions
- Corrupt parameters
- **Craft a new instruction**
- Craft multiple instructions (!)

Mechanisms of a low-level fault model: *fetch skips*.

g:	c.addi sp, sp, -16	c.sw ra, 12(sp)
g+4:	call f	(call *cont.*)

S32	~~c.addi a0, a0, 1~~	~~lw ra, 12(sp)~~
g+12:	(lw *cont.*)	addi sp, sp, 16
g+16:	(addi *cont.*)	c.ret

Annoying consequences:

- Skip one instruction
- Skip two instructions
- Corrupt parameters
- Craft a new instruction
- **Craft multiple instructions (!)**

We need to talk about security

... what is it?!

The countermeasure should make the program "secure".

Unresolved open problem

There is no single definition of security. It depends on the application.

Common examples: we may need that, in the event of a fault...

- The program operates as if not faulted
- The attack is detected and reported
- The program does not leak passwords
- ...

What security property can we achieve here?

- We inherently can't prevent the attack altogether.
- Ideally: recovery, clean detection
- Here: denial of exploitation

Fetch skips hardening property

After a fetch skip, the program will stop/crash before the end of the current block.

What security property can we achieve here?

- We inherently can't prevent the attack altogether.
- Ideally: recovery, clean detection
- Here: denial of exploitation

Fetch skips hardening property

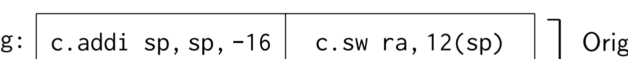
After a fetch skip, the program will stop/crash before the end of the current block.

How?

1. Hardware will compute a checksum of each executed block.
2. Software will compare with expected value.

The countermeasure

The countermeasure: software / hardware opcode checksums.



Original block, except jump

g+12: `call f` | (call *cont.*)

Original jump

The countermeasure: software / hardware opcode checksums.

g:	c.addi sp, sp, -16	c.sw ra, 12(sp)
g+4:	ccscall NEW	*(ccscall cont.)*
g+8:	0x354c	0xc606
g+12:	call f	*(call cont.)*
g+16:	c.ebreak	c.ebreak
g+24:	c.ebreak	c.ebreak

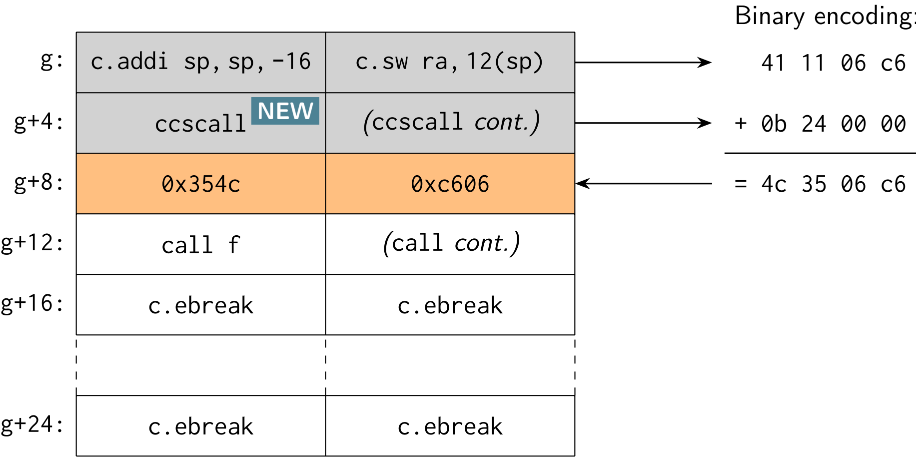
Original block, except jump

Checksum test (needed to jump)
Sum of lines computed by linker.
Exception if mismatch at runtime.

Original jump

Wall of trap instructions
Added by compiler.
Prevents escape from block.

The countermeasure: software / hardware opcode checksums.



The countermeasure: software / hardware opcode checksums.

g:	c.addi sp, sp, -16	c.sw ra, 12(sp)
g+4:	ccscall NEW	*(ccscall cont.)*
g+8:	0x354c	0xc606
g+12:	call f	*(call cont.)*
g+16:	c.ebreak	c.ebreak
g+24:	c.ebreak	c.ebreak

Intuition for security:

Hardware traps on jump unless the previous instruction was ccs/ccscall and it passed.

Too long to be jumped over (12 bytes)

Key design points

[Hardware] RISC-V ISA extension:

- Updates a checksum register for each instruction executed
- One instruction for checksum tests, required before a jump
- As tiny as it gets

[Software] Compiler and linker:

- Provides checksum code and walls
- Linker computes checksums and shuts down two attacks by avoiding values that decode as jumps or checksum instructions

Key design points

[Hardware] RISC-V ISA extension:

- Updates a checksum register for each instruction executed
- One instruction for checksum tests, required before a jump
- As tiny as it gets

[Software] Compiler and linker:

- Provides checksum code and walls
- Linker computes checksums and shuts down two attacks by avoiding values that decode as jumps or checksum instructions NEW

Come on, hardware support?!

This is a reasonable proposition:

- Security *is* in the hardware design process
  - Industrials are seeking ways to improve their hardware
- RISC-V is uniquely extensible/modular
- These are tricky attacks with (currently) no other solutions
  - There's a rich "compromise" space mostly unexplored

Formalizing with semantics

Faulted executions are tricky: formal analysis is a *necessity*.

- Big idea: extend the semantics of assembler! NEW
  - Clears the abstraction gap
  - Pulls in *only* low-level details that are really needed

NOFAULT

$$\frac{}{(\mathrm{PC}, \rho)\ a \Rightarrow [a]\ (\mathrm{PC}, [a])}$$

S32(k)

$$\frac{1 < k \leq N}{(\mathrm{PC}, \rho)\ a \Rightarrow [a + 4k]\ (\mathrm{PC} + 4k, [a + 4k])}$$

S&R32

$$\frac{\rho \neq [a]}{(\mathrm{PC}, \rho)\ a \Rightarrow \rho\ (\mathrm{PC}, [a])}$$

- **Fetch rules** (left): describe the attack
- **Step rules** (below): decoding and execution

ALIGNED-16

$$\frac{\begin{array}{cc}\text{PC aligned} & (\mathrm{PC}, \rho)\ \mathrm{PC} \Rightarrow d\ (\mathrm{PC}_F, \rho') \\ \sigma.\mathrm{CCSDS} = 0 & \mathrm{LSH}(d) \text{ is a 16-bit instruction}^1\end{array}}{\langle \mathrm{PC}, \rho, \sigma, \alpha \rangle \to [\![\mathrm{LSH}(d)]\!](\mathrm{PC}_F, \sigma, \alpha) \bullet \rho'}$$

(...)

Proof of security

Proven security guarantee

If you fetch skip, the program will stop/crash before the end of the current block.
Same for multi-fault attacks (if no checksum collision).

1. Execution only leaves a protected block when the checksum is correct.
   - Cannot reach end of block due to trap wall.
   - Before jumping a ccs is needed, and it can't be forged.
   - The official ccs/jump widget detects attacks.
2. Single-fault case: correct checksum implies no fault.

Note: might crash before end of block.

Implementation

Compilers don't know security (because C doesn't)

```c
int mac(int a, int b, int c) {
  return a + b * c;
}
```

Compiled for RISC-V again:

```
mac:
    mul     a1, a2, a1
    add     a0, a0, a1
    ret
```

- Vulnerable to instruction skip

Compilers don't know security (because C doesn't)

```
mac:
    mul     a3, a2, a1
    mul     a3, a2, a1
    mv      a4, a0
    mv      a4, a0
    add     a0, a4, a3
    add     a0, a4, a3
    ret
    ret
```

- Vulnerable to instruction skip

A countermeasure: duplicate instructions.

- Now secure!

Compilers don't know security (because C doesn't)

```
int mac(int a, int b, int c) {
  int x, y;
  x = b * c;
  x = b * c;
  y = a;
  y = a;
  a = y + x;
  a = y + x;
  return a;
  return a;
}
```

- Vulnerable to instruction skip

A countermeasure: duplicate instructions.

- Now secure!
- Not expressible at source level 🙁

Compilers don't know security (because C doesn't)

```
int mac(int a, int b, int c) {
  int x, y;
  x = b * c;
  x = b * c;
  y = a;
  y = a;
  a = y + x;
  a = y + x;
  return a;
  return a;
}
```

LLVM -O1

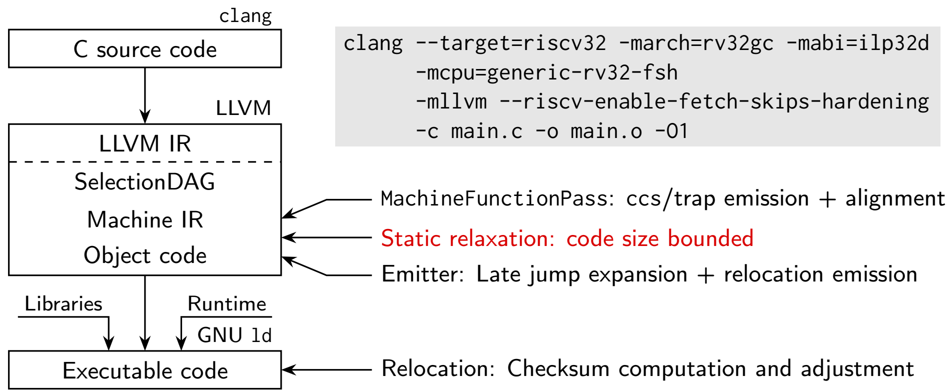
- Vulnerable to instruction skip

A countermeasure: duplicate instructions.

- Now secure!
- Not expressible at source level 😟
- Compiler optimizes it away 😭

```
mac:
    mul     a1, a2, a1
    add     a0, a0, a1
    ret
```

Implementation: a multi-stage process



Experimental validation by simulation

- QEMU support for Xccs and fetch skip injection

```
% qemu-riscv32 -cpu rv32-fsh --riscv-faults '0x40000:s32' ./main
qemu: unhandled CPU exception 0x18 -aborting [...]
```

`0x18`: *Attack detected*

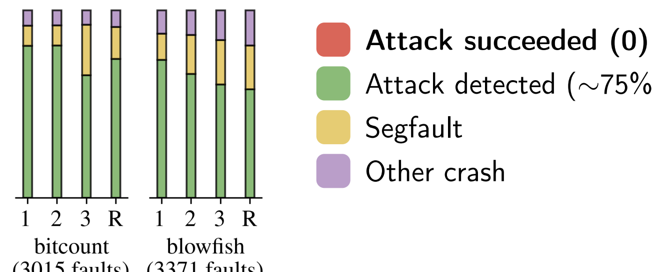
- `-cpu rv32-fsh` selects an Xccs-capable CPU ;
- `-riscv-faults '0x40000:s32'` skips the first 4 bytes.

Experimental validation by simulation

MiBench [Gut+01] benchmarks

1. Exhaustive skip
2. Exhaustive double-skip
3. Exhaustive skip-and-repeat

R. 2000 random multi-faults



- 9 programs, 32’000 attacks reached, 0 bypass (0 checksum collision)
- **Cost: ∼10% time, average x2.46 space** (similar work: x5 time and space)

These are very good because of the software/hardware combo!

Conclusion

Putting it all together

C source code

...

Many compiler IRs

...

ISA/Assembly

Micro-architecture

Gates/RTL

Electrical signals

Putting it all together

C source code
...
Many compiler IRs
...
ISA/Assembly

Micro-architecture	Model: Skip memory fetch	Fault analysis and modeling
	↑	
Gates/RTL	Model: Fail to latch in time	
	↑	
Electrical signals	Model: Glitch clock cycle	

Putting it all together

Level	Property / Model
C source code	Security prop: true
…	
Many compiler IRs	Security prop: (internal definitions)
…	↓
ISA/Assembly	Security prop: stop before end of block

} Security specification, hardening compilation

Level	Model
Micro-architecture	Model: Skip memory fetch
	↑
Gates/RTL	Model: Fail to latch in time
	↑
Electrical signals	Model: Glitch clock cycle

} Fault analysis and modeling

Putting it all together

Level	Property / Model
C source code	Security prop: true
...	
Many compiler IRs	Security prop: (internal definitions)
...	↓
ISA/Assembly	Security prop: stop before end of block
ISA + encoding + fetch	Security proof wrt model
Micro-architecture	Model: Skip memory fetch
	↑
Gates/RTL	Model: Fail to latch in time
	↑
Electrical signals	Model: Glitch clock cycle

Security specification, hardening compilation (C source code to ISA/Assembly)

Fault analysis and modeling (Micro-architecture to Electrical signals)

Understanding and fighting
fault injections
with programming languages

Understanding and fighting **fault injections** with programming languages

Understanding:

- All a matter of crossing abstractions the right way, with formal support

Understanding and fighting
fault injections
with programming languages

Understanding:

- All a matter of crossing abstractions the right way, with formal support

Fighting:

- Software fights for half the abstraction distance with hardening compilation
- Software/hardware combo has a lot to offer
- Deeper toolchain integration needed [Vu21]

Understanding and fighting
fault injections
with programming languages

Understanding:

- All a matter of crossing abstractions the right way, with formal support

Fighting:

- Software fights for half the abstraction distance with hardening compilation
- Software/hardware combo has a lot to offer
- Deeper toolchain integration needed [Vu21]

Thoughts?

References I

[Als+22] Ihab Alshaer et al. “Variable-Length Instruction Set: Feature or Bug?” In: Maspalomas, Spain. IEEE, 2022. ISBN: 978-1-6654-7405-4. DOI: 10.1109/DSD57027.2022.00068.

[Bar+06] H. Bar-El et al. “The Sorcerer’s Apprentice Guide to Fault Attacks”. In: *Proceedings of the IEEE* 94.2 (2006), pp. 370–382. DOI: 10.1109/JPROC.2005.862424.

[Gut+01] M.R. Guthaus et al. “MiBench: A free, commercially representative embedded benchmark suite”. In: Austin, TX, USA. Austin, TX, USA: IEEE, 2001, pp. 3–14. ISBN: 0-7803-7315-4. DOI: 10.1109/WWC.2001.990739.

[Lau20] Johan Laurent. “Modélisation de fautes utilisant la description RTL de microarchitectures pour l’analyse de vulnérabilité conjointe matérielle-logicielle”. Theses. Université Grenoble Alpes, Nov. 2020. URL: https://tel.archives-ouvertes.fr/tel-03167493.

References II

[MDG24] Sébastien Michelland, Christophe Deleuze, and Laure Gonnord. "From Low-Level Fault Modeling (of a Pipeline Attack) to a Proven Hardening Scheme". In: *Proceedings of the 33rd ACM SIGPLAN International Conference on Compiler Construction*. CC 2024. , Edinburgh, United Kingdom, Association for Computing Machinery, 2024, pp. 174–185. ISBN: 9798400705076. DOI: 10.1145/3640537.3641570. URL: https://doi.org/10.1145/3640537.3641570.

[Sol+21] Hadi Soleimany et al. "Practical multiple persistent faults analysis". In: *Cryptology ePrint Archive* (2021).

[Vu21] Son Tuan Vu. "Optimizing Property-Preserving Compilation". 2021SORUS435. PhD thesis. 2021. URL: http://www.theses.fr/2021SORUS435/document.

[Yuc+16] Bilgiday Yuce et al. "Software Fault Resistance is Futile: Effective Single-Glitch Attacks". In: Santa Barbara, CA, USA. Santa Barbara, CA, USA: IEEE, 2016, pp. 47–58. ISBN: 978-1-5090-1109-4. DOI: 10.1109/FDTC.2016.21.