

# Secure compilation

*with the compiler, not against*

---

First experiments on “Tracing LLVM”

Sébastien MICHELLAND (UGA/LCIS, Valence)

PHISIC 2025 — May 20th, 2025



PROGRAMME  
DE RECHERCHE  
CYBERSÉCURITÉ



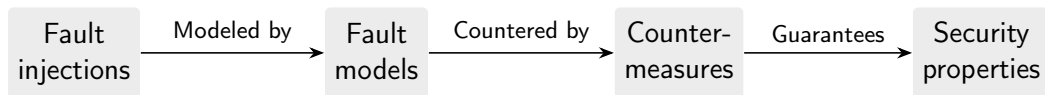
**UGA**  
Université  
Grenoble Alpes

**LCIS**  
Laboratoire de Conception  
et d'Intégration des Systèmes

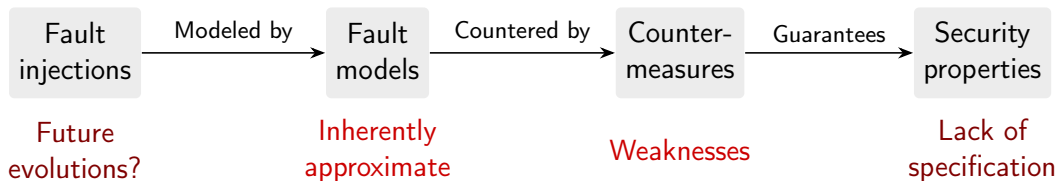
1

# Uncertainties and incompleteness

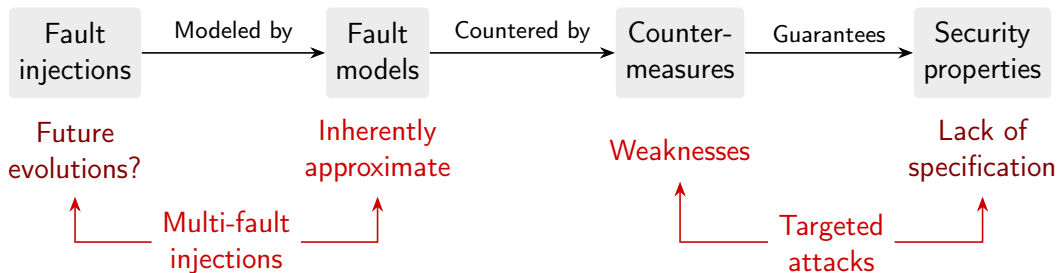
# Fault injections are already way out of the comfort zone



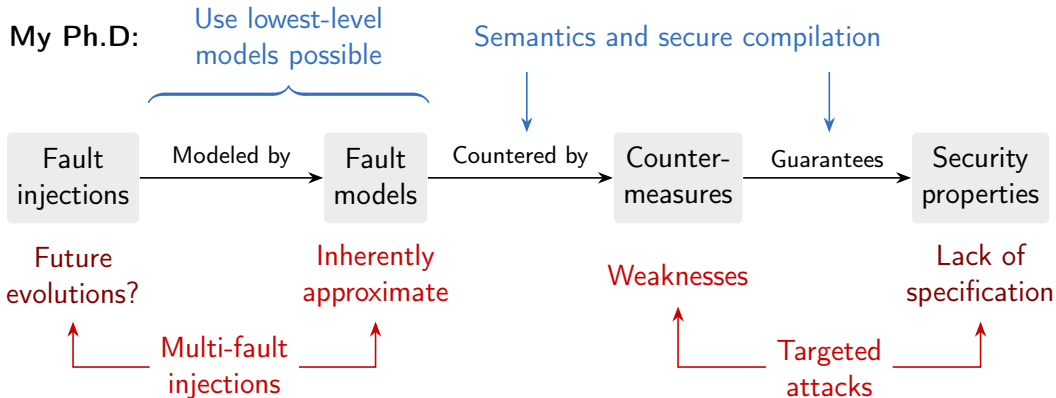
# Fault injections are already way out of the comfort zone



# Fault injections are already way out of the comfort zone



# Fault injections are already way out of the comfort zone



2

# Use lowest-level models possible

## Precise attack models are low-level and tricky



Fetch skips by Alshaer et al. [Als+22]

c.addi a0, a0, 1	lw a0, 144(a1)
(lw cont.)	c.ret

▼ Skip 32 bits!

<del>c.addi a0, a0, 1</del>	<del>lw a0, 144(a1)</del>
addi s2, s2, 1	c.ret

- ▶ Found on ARM and RISC-V
- ▶ Can corrupt instructions
- ▶ Can affect more than one instruction

Typical abstraction compromise!

- ▶ Brings in pipeline details
- ▶ More precise than instruction skip
- ▶ Harder to deal with



# But co-design can deal with them!

Paper: From low-level fault modeling to a proven hardening scheme — CC'24 [MDG24]

✧ Co-designed countermeasure with nice properties!



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

Sébastien Michelland  
sebastien.michelland@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 high-level 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 complex 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 at low-level systems enables finer, more secure countermeasures.

### 1 Introduction

An attacker with access to a physical device can perform fault injection attacks. Physical interference such as a clock glitch, a power supply voltage glitch, or an electromagnetic pulse, can cause hardware to behave erroneously [Bar 02 et al. 2008], sometimes just enough to bypass an application's security. The development of fault injection attacks [Sharp et al. 2021] makes them a tangible threat to modern safety- and security-critical systems. Countering them is uniquely challenging due to the unpredictable effects of low-level interference on high-level security properties – a leap that traditional development tools meticulously avoid by building upon a clear abstraction stack from hardware to programming languages.

In order to conquer the complexity of these attacks, security engineers construct fault models by approximating

faults' effects to a desired level of abstraction. These span from bit flips in RTL, Register Transfer Level latches [Tobias et al. 2022] to failures in pipeline forwarding [Lauront 2020] to corrupted ISA registers [Biarbe et al. 2014] and branch inversion directly in source code [Piot 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 possibly) 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. [Haller et al. 2013]), 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 hardwired in e.g. the hardware). Hardware protections [Schumacher et al. 2018] are less common, but better equipped to deal with local and remote side-channel attacks [Vilho et al. 2007], which share many aspects with fault attacks (see G. [Wideman et al. 2023]).

The key issue with ISA-level fault models is that the approximation is quite crude. [Lauront et al. 2018] shows that faulted behaviors often depend on micro-architectural features and cannot be described accurately without including hardware details. Pipeline analysis in [Vive et al. 2014] further shows that targeted fault attacks can and do defeat many ISA-level countermeasures by exploiting unintended low-level effects.

Naturally, using low-level models widens the abstraction gap between the attack and the countermeasure (often applied during compilation at an IR or back-end level). This creates a risk that protections could be altered or defeated by the compiler's late stages. These cross-layer concerns (commonly avoided by disabling optimizations or losing security claims on exhaustive injection campaigns) resurface when attempting to formally prove a countermeasure's security.

The issue of proving security for countermeasures at the ISA level or lower has received little attention compared to traditional testing. Works that reach previous levels of security

# But co-design can deal with them!

Paper: From low-level fault modeling to a proven hardening scheme — CC'24 [MDG24]

## ✧ Co-designed countermeasure with nice properties!

### ► Simple implementation on both ends

- HW computes checksum of executed opcodes
- SW tests it before every jump

### ► Formalized and proven

- Attacks will crash or be detected quickly

### ► Remarkable performance

- For a strong attacker, 10% time, 2.5x space
- Usual instruction skip CM are 4x time/space

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

Sébastien Michelland  
sebastien.michelland@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 complex 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 system models elicits more secure countermeasures.

##### 1 Introduction

An attacker with access to a physical device can perform fault injection attacks. Physical interference such as a clock glitch, a power supply voltage glitch, or an electromagnetic pulse, can cause hardware to behave erroneously [Bar et al. 2006], sometimes just enough to bypass an application's security. The development of fault injection attacks [Shepard et al. 2021] makes them a tangible threat to modern safety- and security-critical systems. Countering them is uniquely challenging due to the unpredictable effects of low-level interference on high-level security properties — a leap that traditional development tools meticulously avoid by building upon a clean abstraction stack from hardware to programming languages.

In order to conquer the complexity of these attacks, security engineers construct fault models by approximating

faults' effects to a desired level of abstraction. These span from bit flips in RTL, Register Transfer Level latches [Tobias et al. 2022] to failures in pipeline forwarding [Lauront 2020] to corrupted ISA registers [Biarbe et al. 2014] and branch inversion directly in source code [Pinto 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. [Haller et al. 2013]), 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 code in e.g. the hardware). Hardware protections [Lachaux et al. 2011] are less common, but better equipped to deal with local and remote side-channel attacks [Vilho et al. 2007], which share many aspects with fault attacks [see G. [Wideman et al. 2012]].

The key issue with ISA-level fault models is that the approximation is quite crude. [Lauront et al. 2019] shows that faulted behaviors often depend on micro-architectural features and cannot be described accurately without including hardware details. Pipeline analysis in [Vive et al. 2014] further shows that targeted fault attacks can do defeat many ISA-level countermeasures by exploiting unmodeled low-level effects.

Naturally, using low-level models widens the abstraction gap between the attack and the countermeasure (often applied during compilation at an IR or back-end level). This creates a risk that protections could be altered or defeated by the compiler's late stages. These cross-layer concerns (commonly avoided by disabling optimizations or losing security claims on exhaustive injection campaigns) resurface when attempting to formally prove a countermeasure's security.

The issue of proving security for countermeasures at the ISA level or lower has received little attention compared to traditional testing. Works that reach proven levels of security



## Still, we can't be just low-level.

The security property is just “normal behavior or exception”.

- ▶ What about denial of service? Real-time violations? Data leaks?
- ▶ Also not everything needs to be protected...

### Requirement:

- ▶ Source should be able to provide security annotations.

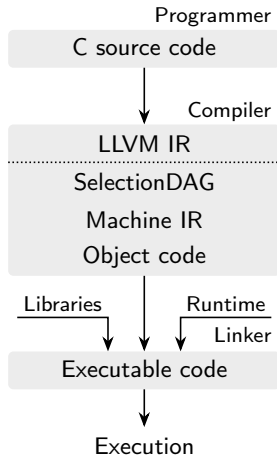
### Often missing at the SW/HW interface

- ▶ Most hardware countermeasures against faults only do functionality
- ▶ Also a social problem!

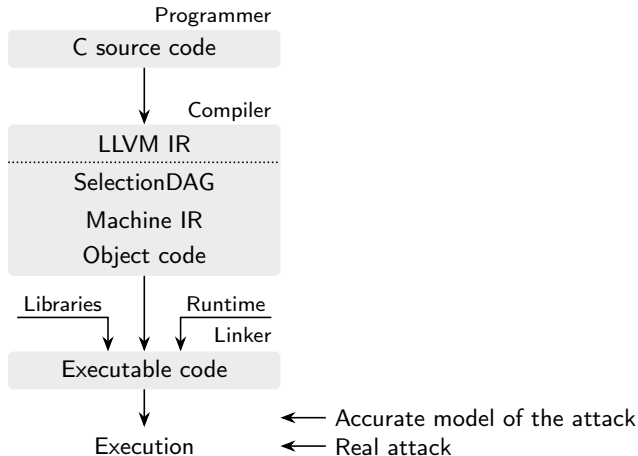
3

# Semantics and secure compilation

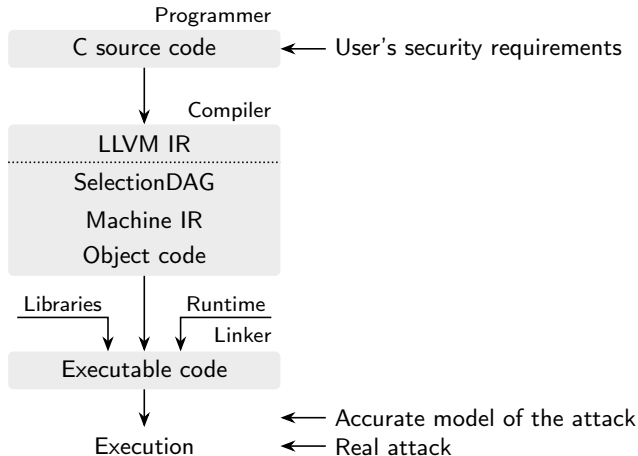
# There is an abstraction gap between attacks and requirements...



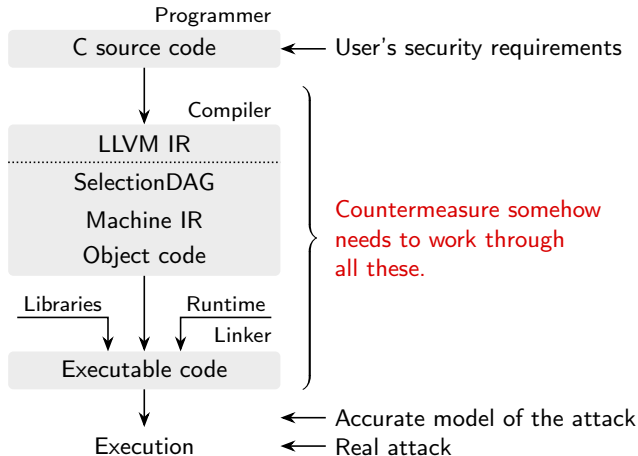
# There is an abstraction gap between attacks and requirements...



# There is an abstraction gap between attacks and requirements...



# There is an abstraction gap between attacks and requirements...





... which only the compiler can properly deal with.

Typically:

- ▶ Harden everything; no control from source code like annotations
- ▶ Harden close to source; no control of assembly (and pray for -O0 to work)
- ▶ Tricks to avoid breakage: volatile abuse, inline assembly, disable passes...

Glaringly insufficient: subtle bugs, no formal guarantees, always a pain.

✨ **Tracing LLVM:** extension of LLVM, currently focused on RISC-V

- ▶ Adds semantic tools that preserve and *trace* elements of the program
- ▶ (Ongoing) Provides an API for querying and accessing traced objects
- ▶ Is intended to be used as a “countermeasure toolbox”

## Tracing demo #1: types

Types capture *data-flow* and are very strong in C and LLVM IR!

```
unsigned char ! __attribute__((trace(dataflow))) cardPin[4];
```

## Tracing demo #1: types

Types capture *data-flow* and are very strong in C and LLVM IR!

```
unsigned char ! __attribute__((trace(dataflow))) cardPin[4];
```

- ▶ *Traced type constructor* “T!”—*secretly* the identity
- ▶ Here we trace the downstream dataflow of cardPin

## Tracing demo #1: types

Types capture *data-flow* and are very strong in C and LLVM IR!

```
unsigned char ! __attribute__((trace(dataflow))) cardPin[4];
```

- ▶ *Traced type constructor* “T!”—*secretly* the identity
- ▶ Here we trace the downstream dataflow of cardPin

### What does this do?

- ▶ Taint expressions that depend on cardPin in the front-end
- ▶ Generate code that can't be rewritten without explicit approval
- ▶ Tag until Machine IR, where we can cleanup all relevant registers

## Tracing demo #2: wrappers

```
int ! __attribute__((trace(writes))) valid;  
valid = FALSE;
```

- ▶ Traces writes to valid, requiring that they occur exactly as written

## Tracing demo #2: wrappers

```
int ! __attribute__((trace(writes))) valid;  
valid = FALSE;
```

- Traces writes to valid, requiring that they occur exactly as written

LLVM IR

```
simplewrapper void 1 2 closed ; hides the store  
store i32 85, ptr %valid
```

## Tracing demo #2: wrappers

```
int ! __attribute__((trace(writes))) valid;  
valid = FALSE;
```

- Traces writes to valid, requiring that they occur exactly as written

LLVM IR

```
simplewrapper void 1 2 closed ; hides the store  
store i32 85, ptr %valid
```

RISC-V Assembler

```
li a0, 85
```

- ... and optimizations cannot touch this even if we enable them!  
\* *except some late back-end locations where there are no wrappers*

# Getting strong countermeasures from tracing

(work currently under review)

I use Tracing LLVM to build a secure `verifyPIN` function with:

- ▶ Basic data-flow integrity (double loads)
- ▶ Basic control-flow integrity (Step Counter Incrementation)
- ▶ All sensitive data allocated in registers
- ▶ Sensitive registers zeroed at exit of function



# Getting strong countermeasures from tracing

(work currently under review)

I use Tracing LLVM to build a secure `verifyPIN` function with:

- ▶ Basic data-flow integrity (double loads) → Source
- ▶ Basic control-flow integrity (Step Counter Incrementation) → Source
- ▶ All sensitive data allocated in registers → Assembly
- ▶ Sensitive registers zeroed at exit of function → Assembly

✨ Can have both source annotations and precise assembly code!

4

# Conclusion

---

## Secure compilation:

*with the compiler, not against*

---

---

## Secure compilation:

*with the compiler, not against*

---

### My contributions

1. Fetch skips countermeasure: software can help with microarch attacks!
2. Tracing LLVM: tools and compilation guarantees for writing countermeasures.

---

## Secure compilation:

*with the compiler, not against*

---

### My contributions

1. Fetch skips countermeasure: software can help with microarch attacks!
2. Tracing LLVM: tools and compilation guarantees for writing countermeasures.

### Take-away messages!

- ▶ Use the *compiler* to connect high-level requirements to low-level secure code
- ▶ Position: we should also do that with SW/HW co-design!

---

## Secure compilation:

*with the compiler, not against*

---

### My contributions

1. Fetch skips countermeasure: software can help with microarch attacks!
2. Tracing LLVM: tools and compilation guarantees for writing countermeasures.

### Take-away messages!

- ▶ Use the *compiler* to connect high-level requirements to low-level secure code
- ▶ Position: we should also do that with SW/HW co-design!

*Questions?*

## Related work

- ▶ Son Tuan Vu's Ph.D [Vu21] (with Karine Heydemann)  
*much of the same pitch, but only preserves passive observations—within the semantics*
- ▶ The Correctness-Security Gap in Compiler Optimization [DPS15] (2015);  
What You Get is What You C [SCA18] (2018)  
*earlier dives into the fundamental challenges in secure compilation*
- ▶ CompaSeC [Gei+23] (a combined control- and data-flow protection)  
*showcases how hard it is to compose countermeasures, thus the need to prove*

# References I

- [Als+22] Ihab Alshaer et al. “Variable-Length Instruction Set: Feature or Bug?” In: *2022 25th Euromicro Conference on Digital System Design (DSD)*. Maspalomas, Spain. IEEE, 2022. ISBN: 978-1-6654-7405-4. DOI: 10.1109/DSD57027.2022.00068.
- [DPS15] Vijay D'Silva, Mathias Payer, and Dawn Song. “The Correctness-Security Gap in Compiler Optimization”. In: *2015 IEEE Security and Privacy Workshops*. 2015, pp. 73–87. DOI: 10.1109/SPW.2015.33.
- [Gei+23] Johannes Geier et al. “CompaSeC: A Compiler-Assisted Security Countermeasure to Address Instruction Skip Fault Attacks on RISC-V”. In: *Proceedings of the 28th Asia and South Pacific Design Automation Conference*. ASPDAC '23. Tokyo, Japan: Association for Computing Machinery, Jan. 2023, pp. 676–682. ISBN: 9781450397834. DOI: 10.1145/3566097.3567925. URL: <https://doi.org/10.1145/3566097.3567925>.



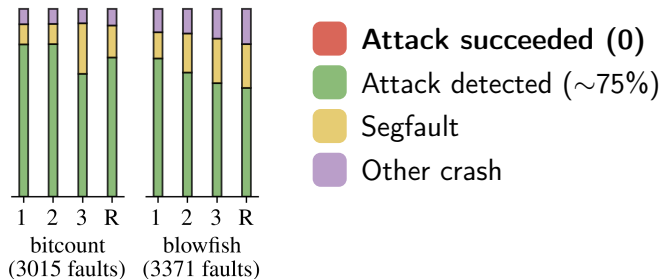
## References II

- [MDG24] Sébastien Michelland, Christophe Deleuze, and Laure Gonnord. “From low-level fault modeling (of a pipeline attack) to a proven hardening scheme”. In: *Compiler Construction (CC’24)*. Edinburgh (Scotland), United Kingdom, Mar. 2024. DOI: 10.1145/3640537.3641570. URL: <https://hal.science/hal-04438994>.
- [SCA18] Laurent Simon, David Chisnall, and Ross Anderson. “What You Get is What You C: Controlling Side Effects in Mainstream C Compilers”. In: *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2018, pp. 1–15. DOI: 10.1109/EuroSP.2018.00009.
- [Vu21] Son Tuan Vu. “Optimizing Property-Preserving Compilation”. Thèse de doctorat dirigée par Heydemann, Karine et Cohen, Albert Henri Informatique Sorbonne université 2021. PhD thesis. Sorbonne Université, 2021. URL: <http://www.theses.fr/2021SORUS435>.

# Fetch skips hardening: validation

MiBench benchmarks

1. Exhaustive skip
2. Exhaustive double-skip
3. Exhaustive skip-and-repeat
- R. 2000 random multi-faults

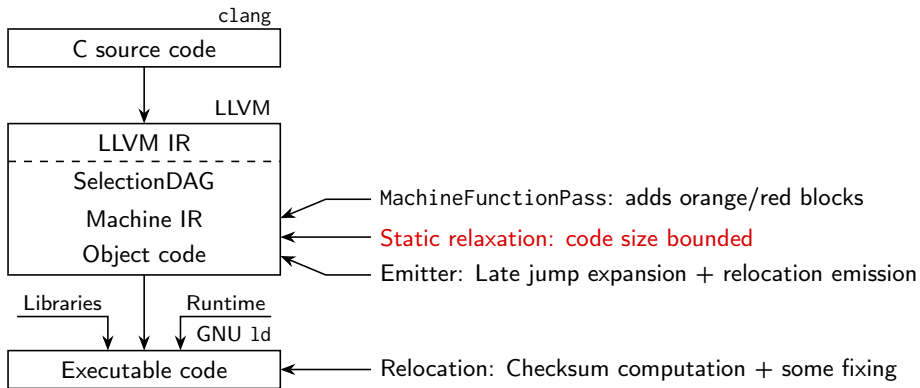


- ▶ 9 programs, 32'000 attacks reached, 0 bypass (0 checksum collision)
- ▶ **Cost: ~10% time, average x2.46 space** (similar work: x5 time and space)

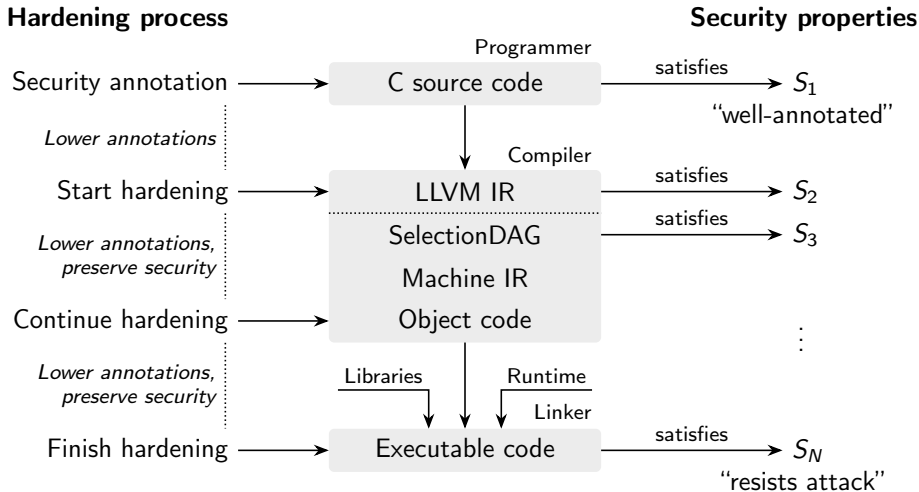
These are very good because of the software/hardware combo!

## Fetch skips hardening implementation

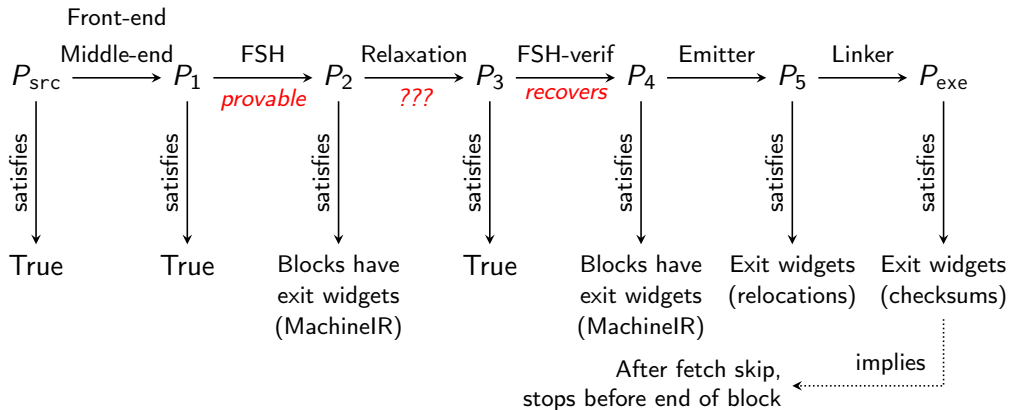
- Fetch Skips Hardening is presented as an assembly transform, but...



# Model of multi-pass hardening



# Security properties of fetch skips hardening



- Almost never talks about fetch skips.

... leading to some of the most robust guarantees

- ▶ To reason about the attack, **extend the semantics of assembler!**
  - ▶ Describe how fetches work to clear the abstraction gap

- ▶ **Fetch rules** (right): describe fetches + attacks
- ▶ **Step rules** (not shown): decoding/execution

### Proven security guarantee

If you fetch skip, the program will stop/crash before the end of the current block.

Multi-fault attacks too (unless checksum collision—usually impossible).

NOFAULT

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

S32(k)

$$1 < k \leq N$$

$$\frac{}{(PC, \rho) \ a \Rightarrow [a + 4k] \ (PC + 4k, [a + 4k])}$$

S&R32

$$\rho \neq [a]$$

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