Secure compilation

with the compiler, not *against*

First experiments on “Tracing LLVM”

Sébastien MICHELLAND (UGA/LCIS, Valence)

Journées Nationales du GDR Sécurité 2025 — June 24th, 2025

FRANCE 2030 PROGRAMME DE RECHERCHE CYBERSÉCURITÉ

UGA Université Grenoble Alpes

LCIS Laboratoire de Conception et d'Intégration des Systèmes

Your last speaker of the day!

Sébastien Michelland

- 3rd-year Ph.D student at LCIS (Valence)
  - Advised by Laure Gonnord and Christophe Deleuze
- Compilers
- Embedded systems
- Formal verification (* rainy days only)

1

Dealing with fault injection attacks

Fault injections are wild...

Fault: abnormal condition leading to incorrect behavior

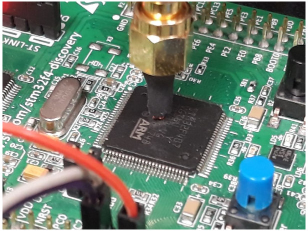
$$2 + 2 \rightarrow \text{something goes wrong} \rightarrow 42$$

Fault injections are wild...

Fault: abnormal condition leading to incorrect behavior

$$2 + 2 \rightarrow \text{something goes wrong} \rightarrow 42$$

Fault injection: creating a fault on purpose



Electromagnetic fault injection [Sol+21]

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

⚠ **Challenges**

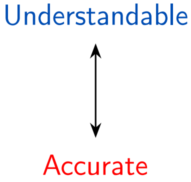
- Can hardly predict outcomes
- Some consistent behaviors
- Many very rare and very weird behaviors

... so we approximate with 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*

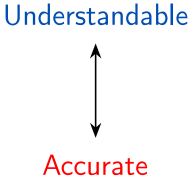


… so we approximate with fault models.

Fault model: approximate description of common fault behaviors

Examples:

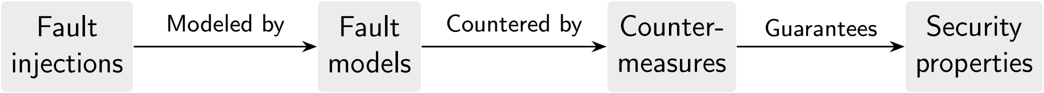
- Invert an `if()` ◀ *C source*
- Corrupt program values ◀ *IR-ish*
- Skip instructions ◀ *Assembly*
- Cancel pipeline forwarding [Lau20] ◀ *Micro-arch*



Inherent tension

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

Getting countermeasures consistent is hard!


Fault injections

Modeled by

Fault models

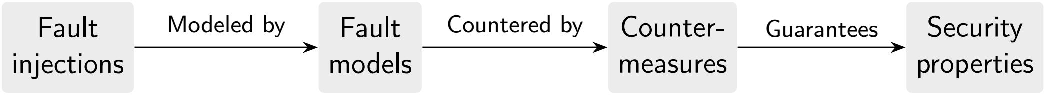
Countered by

Counter-measures

Guarantees

Security properties

Getting countermeasures consistent is hard!



Fault injections → Modeled by → Fault models → Countered by → Counter-measures → Guarantees → Security properties

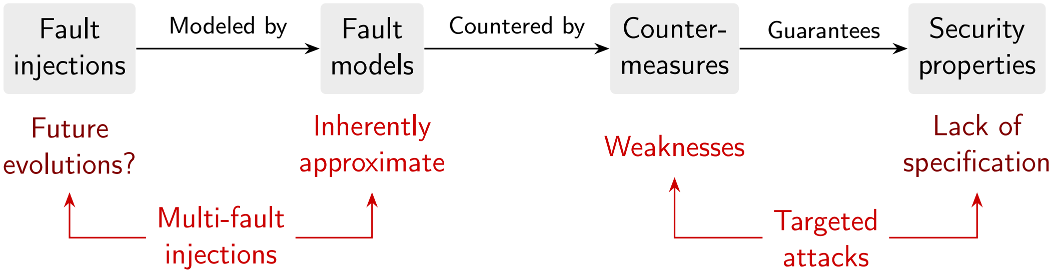
Future evolutions?

Inherently approximate

Weaknesses

Lack of specification

Getting countermeasures consistent is hard!



Fault injections

Modeled by

Fault models

Countered by

Counter-measures

Guarantees

Security properties

Future evolutions?

Inherently approximate

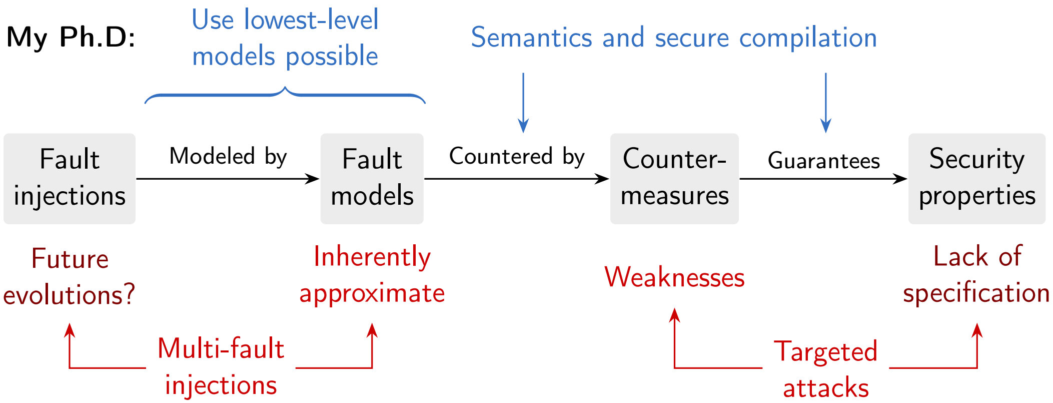
Weaknesses

Lack of specification

Multi-fault injections

Targeted attacks

Getting countermeasures consistent is hard!



My Ph.D:

Use lowest-level models possible

Semantics and secure compilation

Fault injections → Modeled by → Fault models → Countered by → Counter-measures → Guarantees → Security properties

Future evolutions?

Inherently approximate

Weaknesses

Lack of specification

Multi-fault injections

Targeted attacks

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!

~~`c.addi a0, a0, 1`~~	~~`lw a0, 144(a1)`~~
`addi s2, s2, 1`	`c.ret`

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

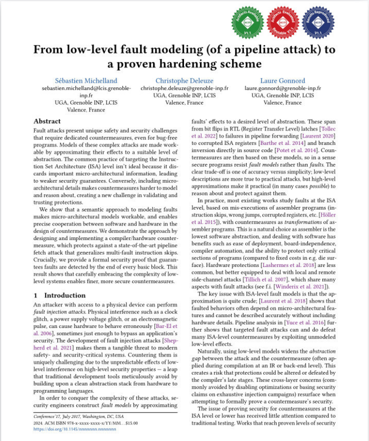
Typical abstraction compromise!

- 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@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.

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-El et al. 2006], sometimes just enough to bypass an application's security. The development of fault injection attacks [Shepherd 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 [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 aspects with fault attacks (see f.i. [Winderix et al. 2021]).

The key issue with ISA-level fault models is that the approximation is quite crude; [Laurent 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 [Yuce et al. 2016] further shows that targeted fault attacks can and 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 basing 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

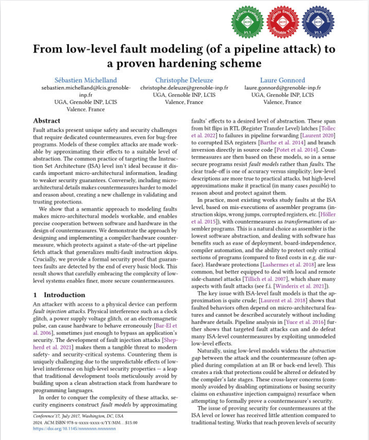
Conference'17, July 2017, Washington, DC, USA
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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
- **Reasonable 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@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.

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-El et al. 2006], sometimes just enough to bypass an application's security. The development of fault injection attacks [Shepherd 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 [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 aspects with fault attacks (see f.i. [Winderix et al. 2021]).

The key issue with ISA-level fault models is that the approximation is quite crude; [Laurent 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 [Yuce et al. 2016] further shows that targeted fault attacks can and 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 basing 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

Conference'17, July 2017, Washington, DC, USA
2024. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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.

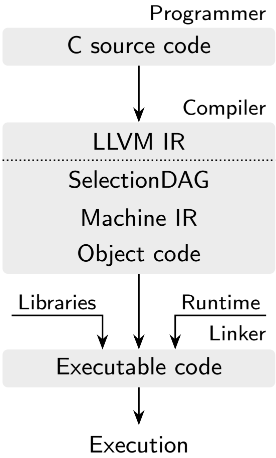
Often missing at the SW/HW interface

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

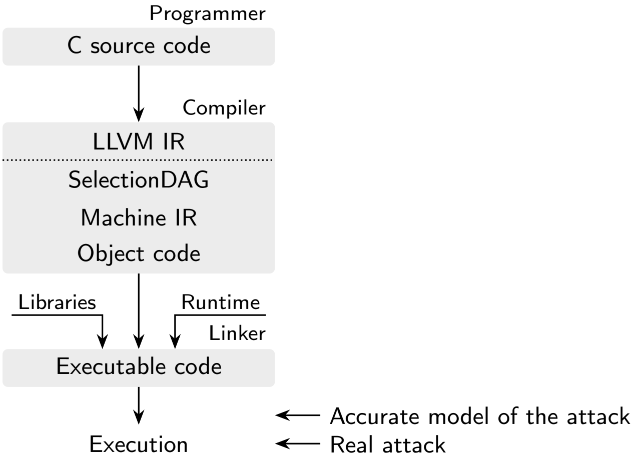
3

Semantics and secure compilation

There is an abstraction gap between attacks and requirements…



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



Programmer

C source code

Compiler

LLVM IR

SelectionDAG

Machine IR

Object code

Libraries

Runtime

Linker

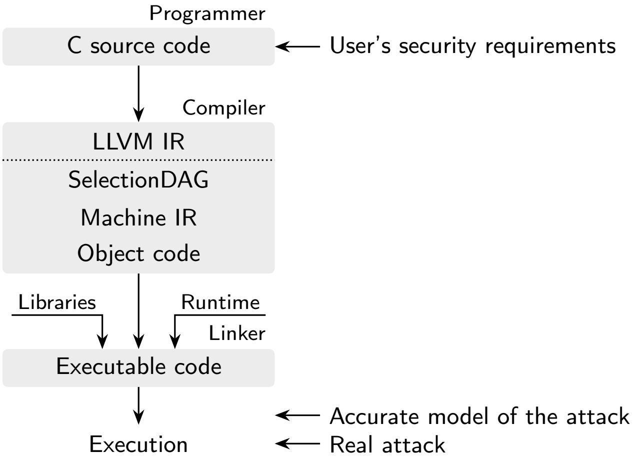
Executable code

Execution

Accurate model of the attack

Real attack

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



Programmer

C source code ← User’s security requirements

Compiler

LLVM IR

SelectionDAG

Machine IR

Object code

Libraries

Runtime

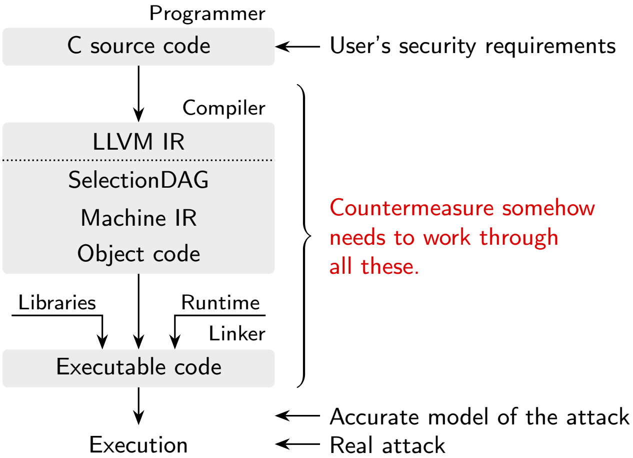
Linker

Executable code

← Accurate model of the attack

Execution ← Real attack

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



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

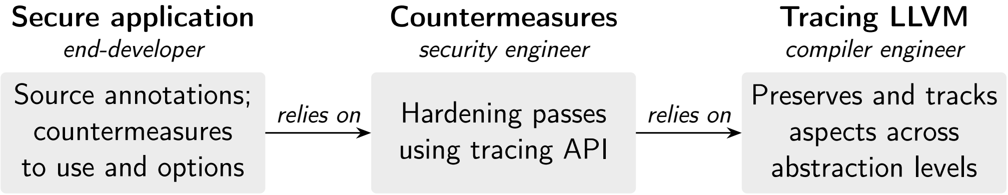
Typically:

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

```
// can you see what's wrong with this?
void *(* volatile memset_ptr)(void *, int, size_t) = &memset;
memset_ptr(array, 0, sizeof array);
```

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

Let’s make the compiler a first-class objective.



Secure application
end-developer

Source annotations; countermeasures to use and options

relies on

Countermeasures
security engineer

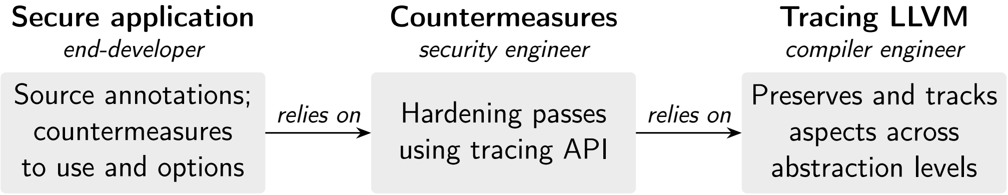
Hardening passes using tracing API

relies on

Tracing LLVM
compiler engineer

Preserves and tracks aspects across abstraction levels

Let’s make the compiler a first-class objective.



Secure application
end-developer
Source annotations; countermeasures to use and options

relies on

Countermeasures
security engineer
Hardening passes using tracing API

relies on

Tracing LLVM
compiler engineer
Preserves and tracks aspects across abstraction levels

✨ **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”

Open-source at https://gricad-gitlab.univ-grenoble-alpes.fr/tracing-llvm

Tracing demo #1: types

We want to cleanup all registers containing data related to `cardPin` when returning.

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

Tracing demo #1: types

We want to cleanup all registers containing data related to `cardPin` when returning.

```
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

We want to cleanup all registers containing data related to `cardPin` when returning.

```
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
- Prevent the compiler from rewriting the computations
- Trace them until Machine IR, where we can cleanup all relevant registers

Tracing demo #2: wrappers

We want to use hardened booleans (`0x55`/`0xaa`) and not have them optimized away.

```
int ! __attribute__((trace(writes))) valid = 0x55;
//  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

- Traces writes to `valid`, requiring that they occur exactly as written in source
- Compiler can't change the values even if it recognizes a boolean

Tracing demo #2: wrappers

We want to use hardened booleans (`0x55`/`0xaa`) and not have them optimized away.

```
int ! __attribute__((trace(writes))) valid = 0x55;
//  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
```

- Traces writes to `valid`, requiring that they occur exactly as written in source
- Compiler can't change the values even if it recognizes a boolean

LLVM IR

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

RISC-V Assembler

```
; unoptimized
li a0, 0x55
```

- ... optimizations can't touch wrappers because it would be incorrect *functionally*
- As a bonus, guaranteed to use the same register for all writes

Getting strong countermeasures from tracing

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

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!

Other features of interest (WIP)

Control lowerings:

- Preserve accesses (like volatile but with register promotion)
- Lower C variable to single unique register (all live ranges)

Trace source code to target code:

- Erase sensitive registers after function
- Guaranteed correct debug information (up to some optimizations lost)

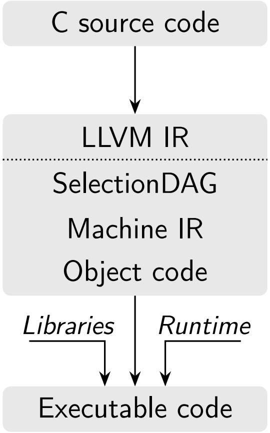
Countermeasure API:

- "What SSA values are `myvar` right now?"
- "Which two assembly xor do my mask refresh?"

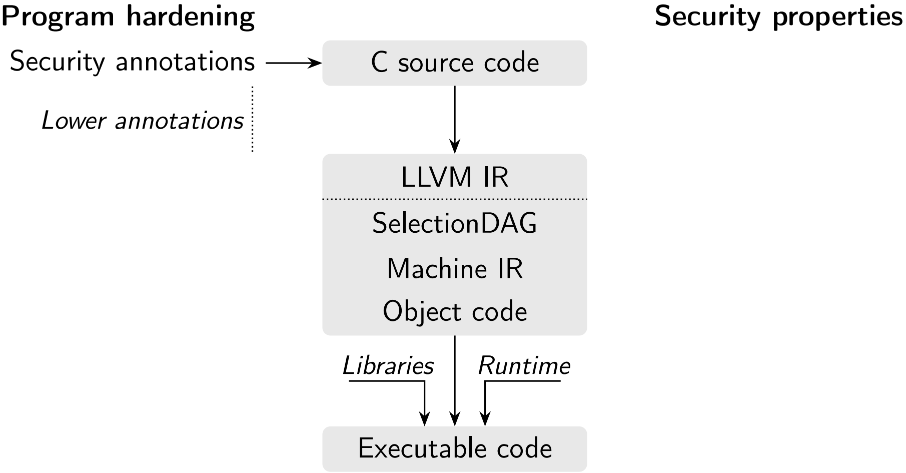
It’s like a parallel compilation!

Program hardening

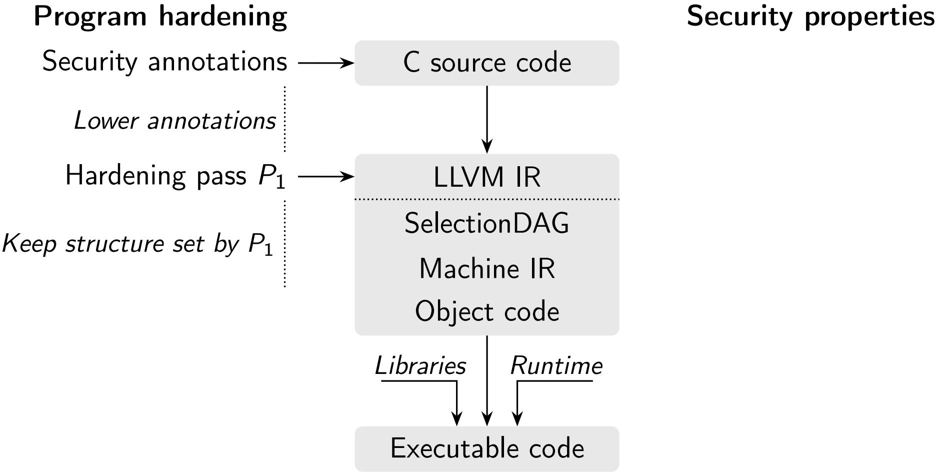
Security properties



It's like a parallel compilation!



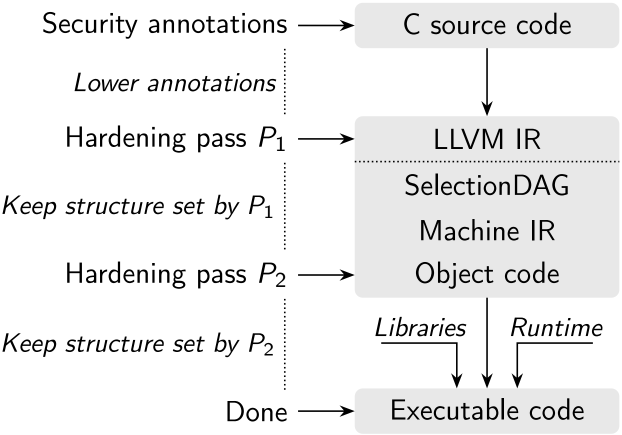
It's like a parallel compilation!



It's like a parallel compilation!

Program hardening

Security properties



Security annotations → C source code

Lower annotations

Hardening pass P_1 → LLVM IR

SelectionDAG

Machine IR

Keep structure set by P_1

Hardening pass P_2 → Object code

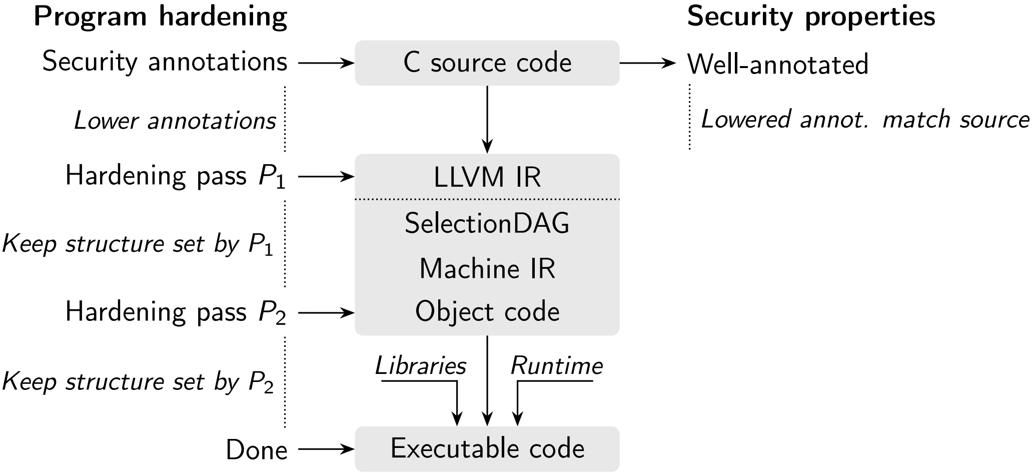
Keep structure set by P_2

Libraries

Runtime

Done → Executable code

It’s like a parallel compilation!



Program hardening

Security annotations → C source code

Lower annotations

Hardening pass P_1 → LLVM IR

SelectionDAG

Keep structure set by P_1

Machine IR

Hardening pass P_2 → Object code

Keep structure set by P_2

Libraries

Runtime

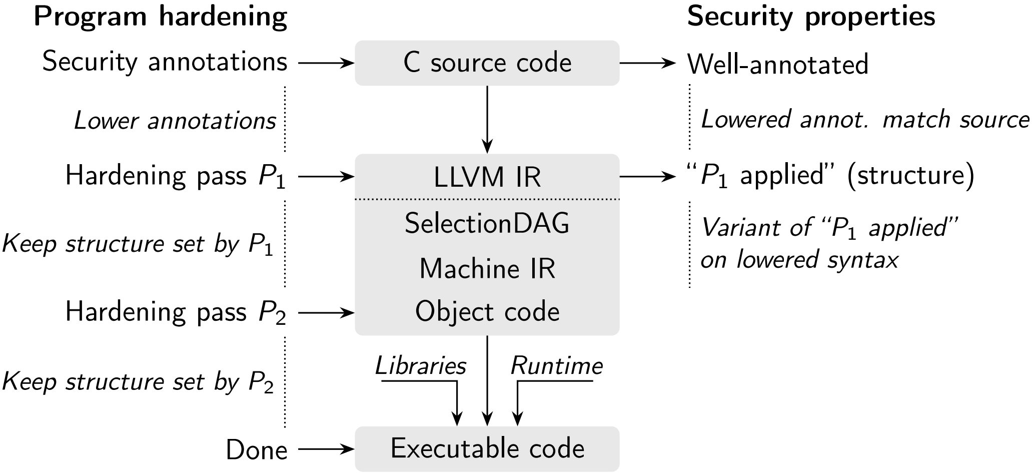
Done → Executable code

Security properties

C source code → Well-annotated

Lowered annot. match source

It's like a parallel compilation!



Program hardening

Security properties

Security annotations → C source code → Well-annotated

Lower annotations

Lowered annot. match source

Hardening pass P_1 → LLVM IR → "P_1 applied" (structure)

SelectionDAG

Keep structure set by P_1

Variant of "P_1 applied" on lowered syntax

Machine IR

Hardening pass P_2 → Object code

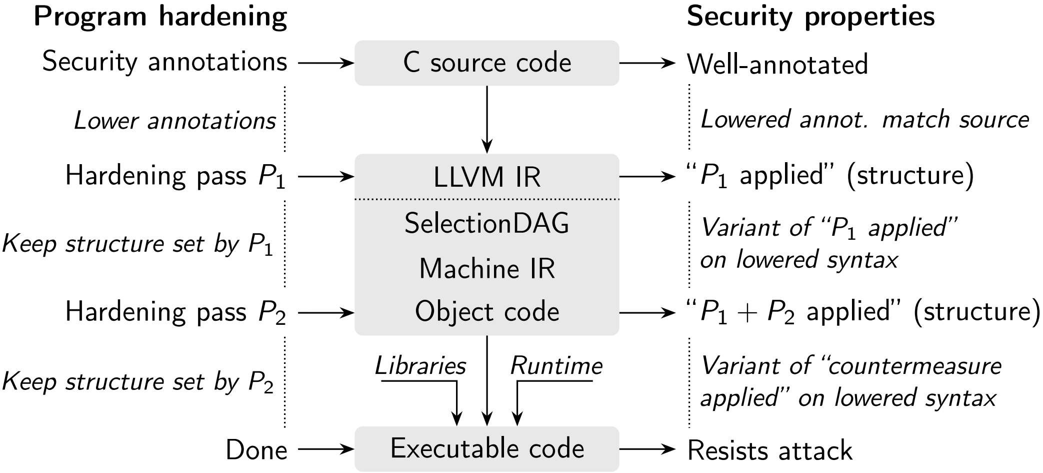
Keep structure set by P_2

Libraries

Runtime

Done → Executable code

It’s like a parallel compilation!



Program hardening

Security annotations

Lower annotations

Hardening pass P_1

Keep structure set by P_1

Hardening pass P_2

Keep structure set by P_2

Done

C source code

LLVM IR

SelectionDAG

Machine IR

Object code

Libraries

Runtime

Executable code

Security properties

Well-annotated

Lowered annot. match source

“P_1 applied” (structure)

Variant of “P_1 applied” on lowered syntax

“$P_1 + P_2$ applied” (structure)

Variant of “countermeasure applied” on lowered syntax

Resists attack

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.
   » `https://gricad-gitlab.univ-grenoble-alpes.fr/tracing-llvm`

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.
   » `https://gricad-gitlab.univ-grenoble-alpes.fr/tracing-llvm`

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.
   » `https://gricad-gitlab.univ-grenoble-alpes.fr/tracing-llvm`

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.

[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: *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`.

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

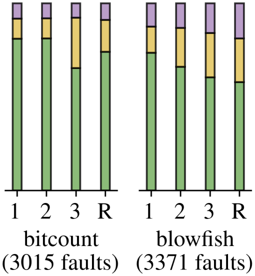
[Vu21] Son Tuan Vu. "Optimizing Property-Preserving Compilation". Thèse de doctorat dirigée par Heydemann, Karine et Cohen, Albert Henri Informatique Sorbonne université 2021. PhD thesis. Sorbonne Université, 2021. URL: `http://www.theses.fr/2021SORUS435`.

Fetch skips hardening: validation

MiBench benchmarks

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

R. 2000 random multi-faults



🟥 **Attack succeeded (0)**

🟩 Attack detected (∼75%)

🟨 Segfault

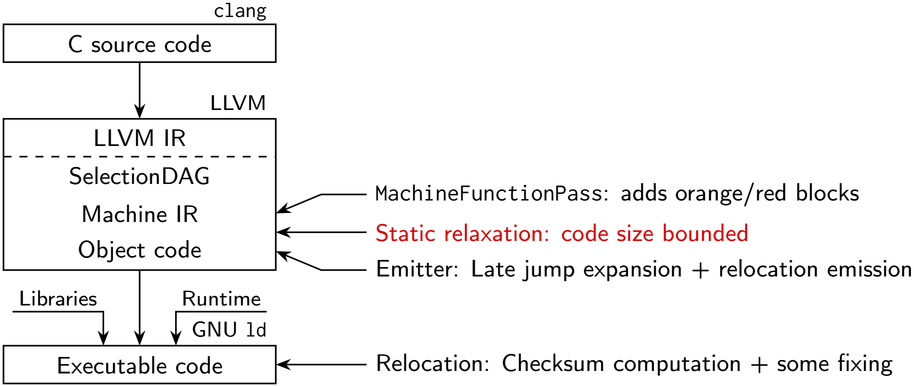
🟪 Other crash

- 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...



clang

C source code

LLVM

LLVM IR

SelectionDAG

Machine IR

Object code

`MachineFunctionPass`: adds orange/red blocks

Static relaxation: code size bounded

Emitter: Late jump expansion + relocation emission

Libraries

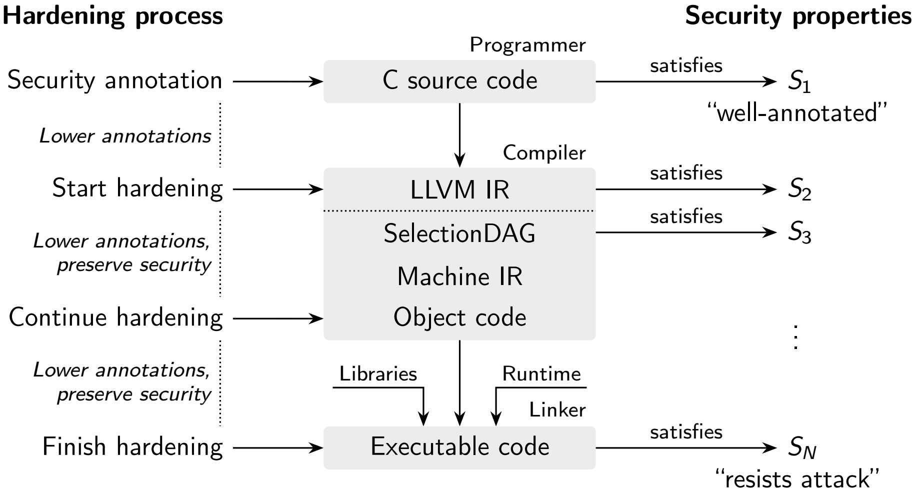
Runtime

GNU `ld`

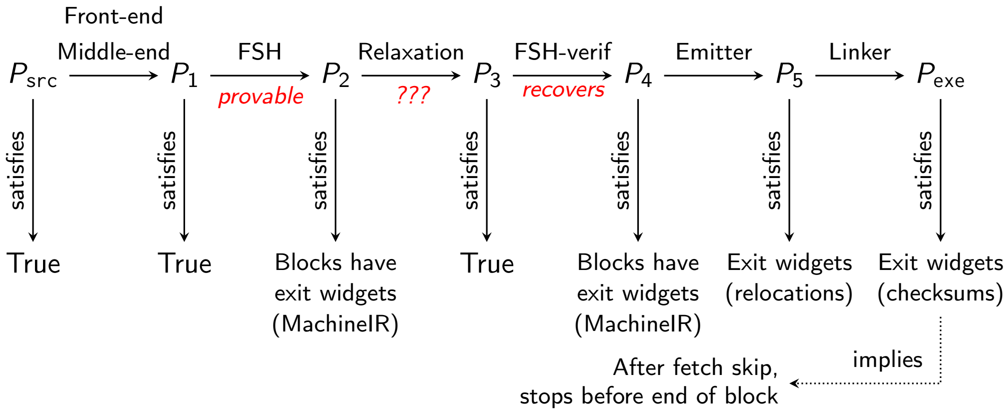
Executable code

Relocation: Checksum computation + some fixing

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).

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

$$\boxed{\mathrm{S32}(k)}\ \frac{1 < k \leq N}{(\mathrm{PC}, \rho)\ a \Rightarrow [a + 4k]\ (\mathrm{PC} + 4k, [a + 4k])}$$

$$\boxed{\mathrm{S\&R32}}\ \frac{\rho \neq [a]}{(\mathrm{PC}, \rho)\ a \Rightarrow \rho\ (\mathrm{PC}, [a])}$$