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

Sébastien MICHELLAND, Christophe DELEUZE, Laure GONNORD

(UGA/LCIS, Valence)

JAIF'24 – October 1st, 2024

FRANCE 2030 PROGRAMME DE RECHERCHE CYBERSÉCURITÉ

anr agence nationale de la recherche

UGA Université Grenoble Alpes

LCIS Laboratoire de Conception et d'Intégration des Systèmes

1

Introduction

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%

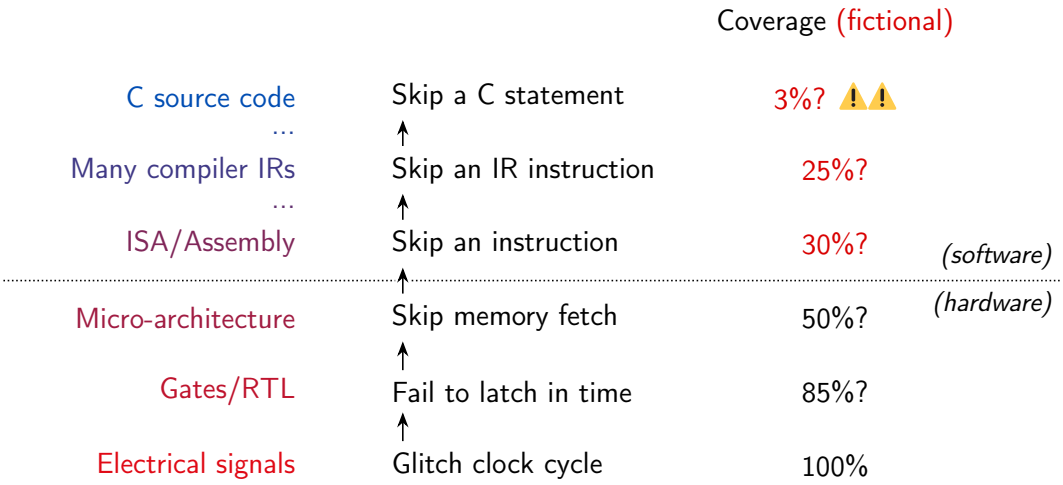
Reversing abstraction descent is *approximate*

		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)*
Micro-architecture			*(hardware)*
Gates/RTL			
Electrical signals			

Reversing abstraction descent is *approximate*



Morality: the cost of modeling at high-level

Approximating undermines security guarantees:

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

In a perfect world... 🌈

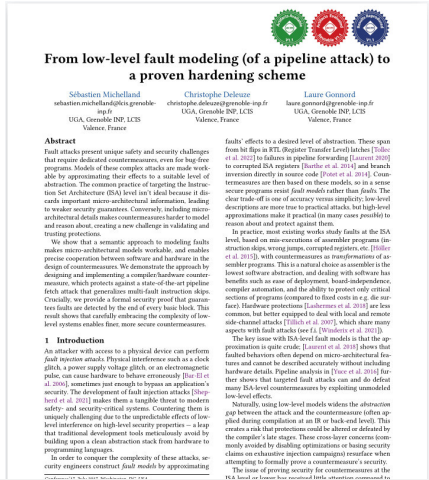
1. Stop fault models' approximations at assembly level or lower
2. Make software generate secure code (tank complexity with semantics)

Reality check: need more theoretical foundations, and old tech (e.g. C) is not helping.

From low-level fault modeling (...) to a proven hardening scheme

Published: CC'24 [MDG24]
https://hal.science/hal-04438994

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



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

Sébastien Michelland
sebastien.michelland@cea.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. 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 costs in e.g. the 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. 2014] further shows that targeted fault attacks can and do defeat many ISA-level countermeasures by exploiting unmodeled low-level effects.

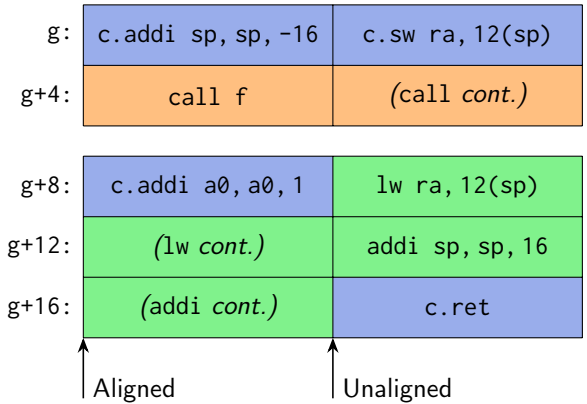
Naturally, using low-level models widens the *abstraction gap* between the attack and the countermeasure (often applied during compilation at an IR or back-end level). This creates a risk that protections could be altered or defeated by the compiler's late stages. These cross-layer concerns (commonly 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

2

Fetch skips

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




```c
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*.

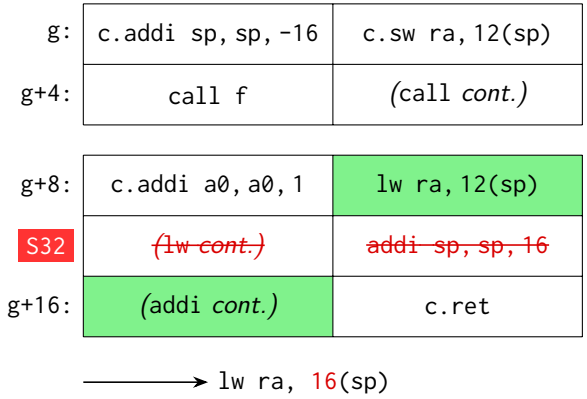
S32	~~`c.addi sp, sp, -16`~~	~~`c.sw ra, 12(sp)`~~
g+4:	`call f`	(`call` *cont.*)

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 (!)

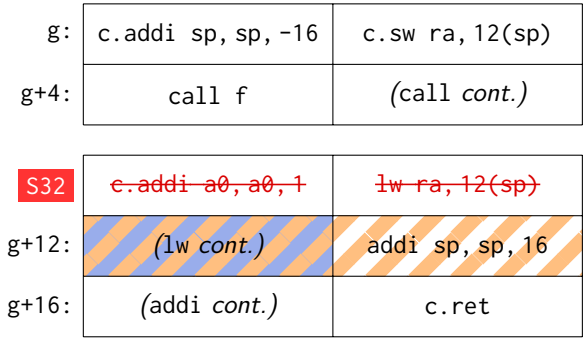
Mechanisms of a low-level fault model: *fetch skips*.



Annoying consequences:

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

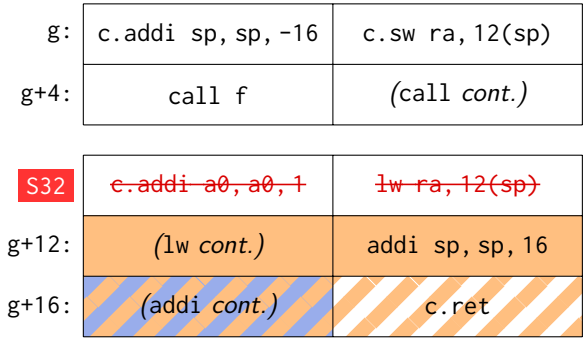
Mechanisms of a low-level fault model: *fetch skips*.



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*.



Annoying consequences:

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

What security property can we achieve here?

- We inherently can't prevent the attack altogether. 🔥
- Ideally: recovery, clean detection
- Here: prevent attacker from exploiting corrupted states

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: prevent attacker from exploiting corrupted states

Fetch skips hardening property

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

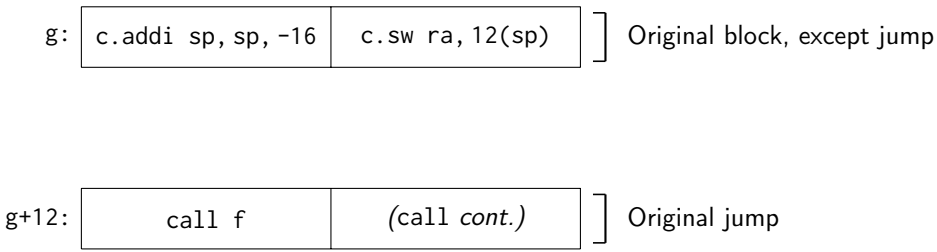
How?

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

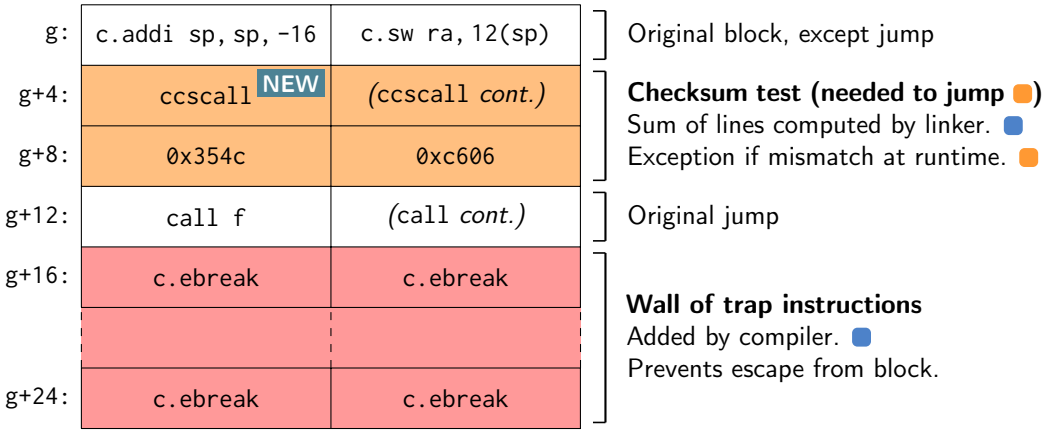
3

The countermeasure

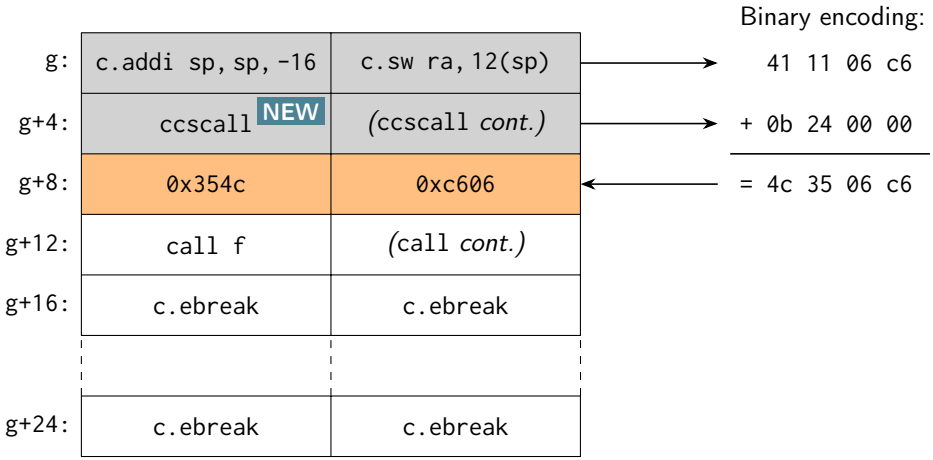
The countermeasure: software / hardware opcode checksums.



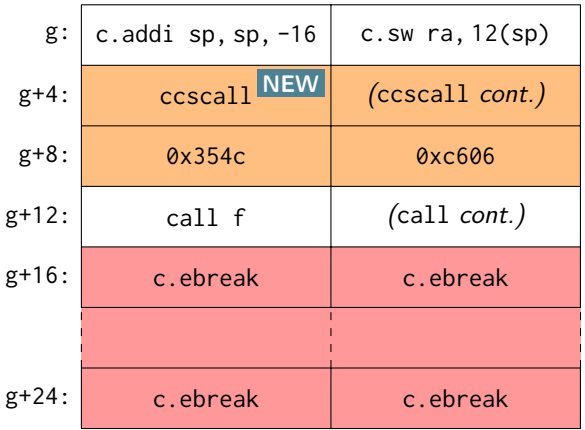
The countermeasure: software / hardware opcode checksums.



The countermeasure: software / hardware opcode checksums.



The countermeasure: software / hardware opcode checksums.



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
- Hardware support reasonable
  - Tiny extension for a modular architecture (RISC-V)
  - No software-only option anyway

[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
- Hardware support reasonable
  - Tiny extension for a modular architecture (RISC-V)
  - No software-only option anyway

[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

Formal semantics and proof of security

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

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

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

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

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

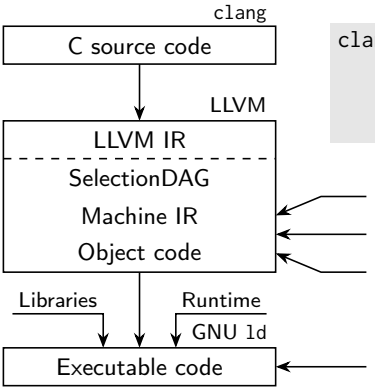
Proven security guarantee

If you fetch skip, the program will stop/crash before the end of the current block.
Same for multi-fault attacks (unless checksum collision–usually impossible).

4

Implementation

Implementation: a multi-stage process




```
clang --target=riscv32 -march=rv32gc -mabi=ilp32d
      -mcpu=generic-rv32-fsh
      -mllvm --riscv-enable-fetch-skips-hardening
      -c main.c -o main.o -O1
```

MachineFunctionPass: ccs/trap emission + alignment

Static relaxation: code size bounded

Emitter: Late jump expansion + relocation emission

Relocation: Checksum computation and adjustment

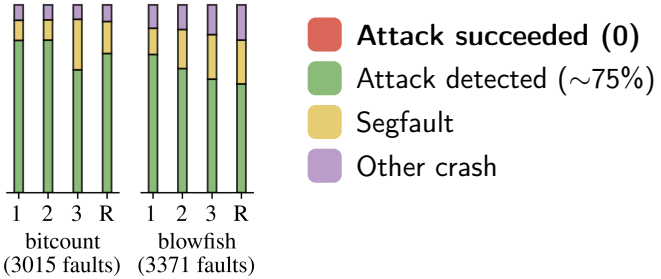
Experimental validation by simulation

- QEMU support for the scheme and for fetch skip injection

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!

5

Conclusion

Conclusion

Spooky low-level attack tackled by software/hardware co-op with formal analysis.

Novelties

- Protect against a microarch-level attack
- Semantic modeling and proof!

Insights and future work

- Crossing the abstraction gap: possible, but rigorously
- Deeper toolchain integration for security passes in compilers (based on [Vu21])

Conclusion

Spooky low-level attack tackled by software/hardware co-op with formal analysis.

Novelties

- Protect against a microarch-level attack
- Semantic modeling and proof!

Insights and future work

- Crossing the abstraction gap: possible, but rigorously
- Deeper toolchain integration for security passes in compilers (based on [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.

[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.

[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.

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

References II

[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.