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

Sébastien Michelland, Christophe Deleuze, Laure Gonnord

▶ **To cite this version:**

Sébastien Michelland, Christophe Deleuze, Laure Gonnord. From low-level fault modeling (of a pipeline attack) to a proven hardening scheme. Compiler Construction (CC'24), Mar 2024, Edinburgh (Scotland), United Kingdom. 10.1145/3640537.3641570 . hal-04438994v1

HAL Id: hal-04438994

https://hal.science/hal-04438994v1

Submitted on 5 Feb 2024 (v1), last revised 20 Feb 2024 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Distributed under a Creative Commons Attribution - ShareAlike 4.0 International License

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

The ARSENE project was funded by the "France 2030" government investment plan managed by the French National Research Agency, under the reference ANR-22-PECY-0004.

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

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 usually focus on specific boards (e.g. RTL model checking in [Tollec et al. 2022]) or high-level languages (e.g. symbolic analysis in [Potet et al. 2014]). Proving traditional countermeasures would provide an appropriate response to [Yuce et al. 2016] by guaranteeing the absence of attack opportunities, across multiple abstraction layers starting at the micro-architectural level.

This approach contrasts with recent progress in working out sophisticated hardening schemes that combine multiple types of protections or resist multiple consecutive or independent attacks (*multi-fault attacks*) [Geier et al. 2023; Werner et al. 2022]. The applicability of these solutions still hinges on ISA-level fault models adequately capturing faults' effects, which is not guaranteed. To ensure security at a lower abstraction level, we choose to refocus on single but accurate models, which we address with a software/hardware co-design approach.

Contribution summary. We present the first formally-proven countermeasure addressing a low-level fault model. We formalize the fault in an extended assembler language in Section 2, then propose in Section 3 a countermeasure based on a hardware extension and a compile/link-time program transformation. From the formal semantics for the fault, we derive a security theorem in Section 4. We implement the scheme in LLVM and GNU `ld`, then evaluate its security with emulated fault campaigns and its performance with processor simulations in Section 5. Finally, we compare this work with recent literature in Section 6, and conclude in Section 7. The supplementary material for this paper contains a detailed operational semantics of the extended assembler and the complete proof of the security theorem.

2 Formal description of fetch skips

We protect against a slightly generalized version of the fault model described by [Alshaer et al. 2022], adapted for RISC-V[1]. The target system is a 32-bit little-endian RISC-V processor with the "C" extension for compressed instructions (such as RV32IC [*The RISC-V Instruction Set Manual Vol. I* 2019]), which means it has a mix of 16-bit and 32-bit instructions.

The instruction set includes the usual arithmetic instructions (`add`, `xor`…), memory instructions (`lw`, `sw`…), conditional (`beq`…) and unconditional (`j`, `jal`…) branches. All instructions have a 32-bit encoding unless prefixed with "`c.`", which denotes a 16-bit encoding from the "C" extension.

For brevity, we use "aligned" to mean a multiple of 4 (bytes) and "unaligned" for a multiple of 4 plus 2. We also use the term "line" to refer to 4 bytes of data at an aligned address.

[1]RISC-V is an open standard ISA; details can be found at https://riscv.org.

```
int g(int x) { return f(x) + 1; }
g:
  # ▾ Push register ra to stack
  24: 41 11       c.addi  sp, sp, -16
  26: 06 c6       c.sw    ra, 12(sp)
  # ▾ Call f (linker later inserts address of f)
  # ▾ ra is both target address and return address
  28: 97 00 00 00   auipc   ra, 0
  2c: e7 80 00 00   jalr    ra, 0(ra)

  # ▾ Add 1 to return value a0 of f
  30: 05 05       c.addi  a0, a0, 1
  # ▾ Pop ra from stack and return a0
  32: b2 40       c.lw    ra, 12(sp)
  34: 41 01       c.addi  sp, sp, 16
  36: 82 80       c.ret
```

24:	`c.addi`	`c.sw`
28:	`auipc` (1/2)	`auipc` (2/2)
2c:	`jalr` (1/2)	`jalr` (2/2)

30:	`c.addi`	`c.lw`
34:	`c.addi`	`c.ret`

Figure 1. C code, object code, and memory layout of a simple function `g(x) = f(x) + 1`.

2.1 RISC-V programs and their execution

This section introduces the notions needed to describe the fault and state the security theorem in Section 4; the full formal semantics can be found in the supplementary material.

We model RISC-V programs as collections of *blocks* made of non- or conditionally-branching instructions, terminated by an unconditional branch.[2] Figure 1 shows a function `g(x) = f(x) + 1` consisting of two blocks, one that calls `f` (implicitly forwarding the argument) and one that increments the return value `a0` then returns.

We model the execution as a sequence of *execution steps* in which the CPU obtains the next instruction (with a combination of a buffer read and/or a memory fetch), decodes it, and executes it. For instance, function `g` in Figure 1 executes in 8 steps (ignoring the call to `f`), with each step running one instruction and consuming either 2 or 4 bytes of code.

The central behavior of interest in this paper is that *there is not always one fetch for one step*: certain instructions are fetched in advance, and others are read by the CPU over two consecutive fetches. This is because most CPUs *always fetch 4 aligned bytes in memory*, that is, a *line* in the memory layout table. For example, the first step in running `g` fetches the 4 bytes at address 24 and puts them in a decoding buffer,

[2]These aren't classical basic blocks; not counting conditional branches as terminators works better for this fault. This is of minor importance except in the security proof.

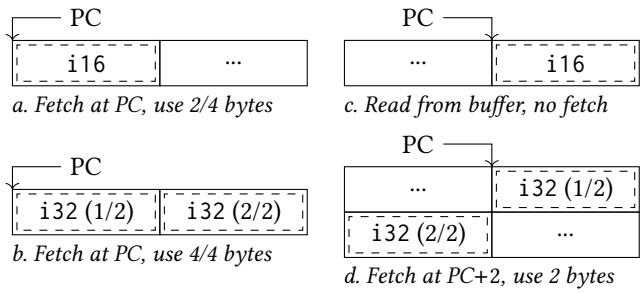

Figure 2. Execution step patterns. `i16`/`i32` *represent arbitrary 16/32-bit instructions. The dashed region is the next instruction to be run.*

but only consumes the first 2 to execute `c.addi`. Then, the second step executes `c.sw` from the buffer without needing another memory fetch.

Figure 2 shows all the ways a step may use buffered data and/or fetch from memory when executing one instruction. When PC is aligned (left), the next instruction is on a new line, so a fetch is performed. When PC is unaligned (right), 2 bytes of instruction data are left in the decoding buffer. If they form a 16-bit instruction (case *c*), the CPU runs it immediately without a fetch. If they form the first half of a 32-bit instruction (case *d*), a fetch is performed at PC + 2 to obtain the second half and piece together the opcode. Each step then increments PC by 2 or 4, setting up the next cycle.

2.2 Fault model

We are interested in protecting programs against *"fetch skip"* attacks described by [Alshaer et al. 2022]. Fetch skips refine the traditional *instruction skip* fault model by analyzing the effect of physical attacks (here, clock glitches) on the micro-architecture rather than on the program only. Because the model is more accurate with respect to physical effects, a countermeasure against fetch skips will be more secure in practice than a countermeasure against plain instruction skip. Insérer le 80% d'Ihab

The different types of fetch skips are illustrated in Figure 3. In this model, attempts by the CPU to fetch at an address L may result in:

- **Skip 32 bits**, k times ($\mathtt{S32}(k)$): Instead of returning the memory contents at address L, the memory contents at address $\mathtt{L}+4k$ are returned, and PC is incremented by $4k$.[3]
- **Skip and repeat 32 bits** (S&R32): The contents at address L are requested from memory, but due to the short cycle the decoding stage triggers before the response is available. A copy of the previous line (usually at $\mathtt{L}-4$) is returned to the decoder for the current step, and the requested value arrives in the decoding buffer later.

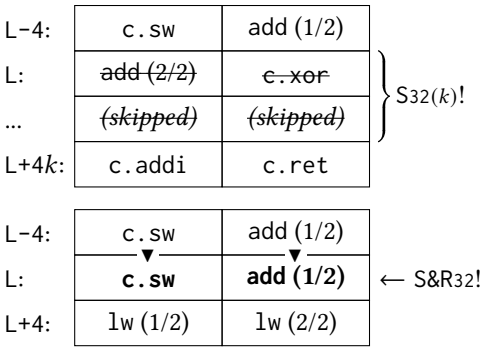

Figure 3. Effect of fetch skip attacks on fetch requests.

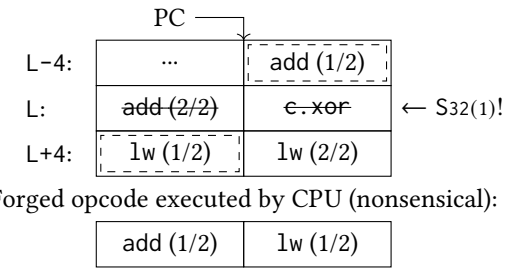

Figure 4. Forging a 32-bit instruction with a $\mathtt{S32}(1)$ attack.

We consider a "level N attacker" that can independently attack every fetch with either one $\mathtt{S32}(k)$ attack ($1 \le k \le N$) or one S&R32 attack.[4]

The connection between fetch skips and the memory layout of instructions creates a new effect, which we call **instruction forging** and illustrate in Figure 4, that cannot be captured by the usual "instruction skip" model. Here, $\mathtt{PC} = \mathtt{L} - 2$ and the decoding buffer contains the first half of a 32-bit unaligned `add` instruction. The CPU fetches at L to obtain the second half (Figure 2, case *d*). Attacking this fetch with $\mathtt{S32}(1)$ results in an unrelated 16-bit value (the start of an `lw`) being used to complete the `add`, causing the execution of an opcode not originally in the program. This opcode can be "anything" (including illegal). Alshaer et al. [2022] demonstrate how this enables new vulnerabilities (e.g. by forging control flow instructions).

Once PC is out of sync, forging can continue without repeated fault injection. Continuing with Figure 4, after running the forged instruction we get $\mathtt{PC} = \mathtt{L} + 6$. Now the second half of `lw` is interpreted as its own 16-bit or (first half of) 32-bit instruction. Thus, the attack carries over to the next line; in the worst case, execution might not resynchronize with the intended sequence of instructions until a jump.

2.3 Program model

To design and prove a countermeasure against such a low-level attack, it is helpful to reflect the faults' effects into

[3] [Alshaer et al. 2022] observes that a the CPU may run a `nop` during a $\mathtt{S32}(k)$ attack; we omit this detail here but account for it in the countermeasure.

[4] We use these patterns because the behavior of more complex combinations cannot be inferred from [Alshaer et al. 2022] without more hardware details.

Name	Rule statement
NoFault	$(\mathrm{PC}, \rho)\ a \Rightarrow [a]\ (\mathrm{PC}, [a])$
S32(k)	$(\mathrm{PC}, \rho)\ a \Rightarrow [a+4k]\ (\mathrm{PC}+4k, [a+4k])$
S&R32	$(\mathrm{PC}, \rho)\ a \Rightarrow \rho\ (\mathrm{PC}, [a])$

Figure 5. Short description of fetch rules.
$(PC, \rho)\ a \Rightarrow d\ (PC', \rho')$ indicates that the fetch of address a with current PC and ρ returns value d to the decoder, along with updated state PC', ρ'. $[a]$ denotes the memory contents at address a.

language features, and study the updated language with semantic tools. In this section, we build a language model that captures the memory layout of instructions while ignoring most concrete effects (apart from jumps and select registers).

Syntax. A *program* is a collection of *blocks*. A block bb is a sequence of *instructions* $\mathtt{bb} = [\mathtt{i}_1, \ldots, \mathtt{i}_{|\mathtt{bb}|}]$ where each instruction $\mathtt{i}_j$ is a 16-bit or 32-bit value, representing an assembler instruction encoded as described by the RISC-V ISA (or, as will be described in Section 3.2, a 32-bit "checksum literal"). We write $\|\mathtt{i}\|$ for the size in bytes of instruction i (which is either 2 or 4).

Program state. Due to the stateful nature of S&R32, which duplicates the "previous" fetched line, a full description of the program state must include the decoding buffer. We model this state with a 4-tuple $\langle \mathrm{PC}, \rho, \sigma, \alpha \rangle$, containing:

- The current PC value (always a multiple of 2);
- ρ, the last 4-byte value fetched by the CPU (decoding buffer). ρ is used to decode at unaligned PCs (Figure 2, right) and replaces the targeted line during a S&R32 attack;
- σ, the values of the Xccs registers associated with our countermeasure, whose role is described in Section 3.2;
- α, the rest of the architectural state (other registers, memory...), which we abstract away as an opaque value.

The attacks shown in Figure 3 are formalized in terms of this state in Figure 5. To illustrate alignment dynamics, attacking the fetch at `0x30` during the execution of `g` (just upon returning from `f`) produces the following execution:

$\langle$`0x30`, $\rho_0, \sigma_0, \alpha_0\rangle$
↓ S32(1): (`0x30`, ρ_0) `0x30` $\Rightarrow$ [`0x34`] (`0x34`, [`0x34`])
↓ Run instruction: `c.addi sp, sp, 16`
$\langle$`0x36`, ρ_1 = [`0x34`], $\sigma_1, \alpha_1\rangle$
↓ No fetch. Run instruction: `c.ret`
$\langle \alpha_1$.ra = `0x30`, $\rho_2 = \rho_1, \sigma_2, \alpha_2\rangle$

`c.ret` jumps to the address stored in register `ra`, which is still `0x30` because we skipped the load from the stack, later leading to a typical stack corruption crash.

3 A co-designed countermeasure

In this section, we propose a software/hardware countermeasure, based on code instrumentation with hardware support.

```
g: # PC >= 0x40000 is a protected region
  400e4: 41 11       c.addi   sp, sp, -1
  400e6: 06 c6       c.sw     ra, 12(sp)
  # ▾ auipc was relocated
  400e8: 97 00 00 00 auipc    ra, 0
  # ▾ Checksum was set and LSB flipped
  400ec: 0b 64 00 00 ccscallb 8                (22,10)
  400f0: e2 75 06 c6 .word    0xc60675e2       (22,11)
  # ▾ jalr was relocated
  400f4: e7 80 00 fb jalr     ra, -80(ra)
  400f8: 02 90       c.ebreak # 8 times        (28)

  40108: 05 05       c.addi   a0, a0, 1
  4010a: b2 40       c.lw     ra, 12(sp)
  4010c: 41 01       c.addi   sp, sp, 16
  4010e: 01 00       c.nop                     (19)
  # ▾ Checksum was set, no flip needed
  40110: 0b 10 00 00 ccs                       (24,13)
  40114: 51 16 b3 40 .word    0x40b31651       (24,14)
  40118: 82 80       c.ret
  4011a: 02 90       c.ebreak # 8 times        (28)
```

Figure 6. Linked object code for g after hardening ($N = 2$).
Numbers on the right refer to lines of Algorithm 1 that added the instructions.

3.1 Overview

The key ideas of the countermeasure are as follows. Machine code is augmented ("hardened") during compilation with checksum protections that react to a fault attack by forcing execution to trap before the end of the current block. This limits exploits by ensuring a sufficiently tight window between attack and detection (like most countermeasures it doesn't prevent side-effects immediately resulting from the fault, which is inherently difficult due to timing). Figure 6 shows the hardened code for the g function from Figure 1.

Hardware is modified to automatically maintain a running checksum (in fact, a simple *sum*) of every line of instruction data fetched from memory during the execution of a block, independent of instruction alignment. Blocks are compiled so that every exit is guarded by a `ccs` instruction (from our ISA extension), which traps if the running checksum is not equal to a reference value computed at compile-time. Blocks thus act as autonomous "jails", in that faulty executions causing the checksum to deviate from its expected value cannot leave their current block.

The co-designed nature of the countermeasure is a result of the cross-checking of information between hardware, whose monitoring produces a trace (checksum) sensitive to fault attacks, and software, which provides reference checksum values to interpret that trace.

Section 3.2 describes the hardware extension that we rely on for the countermeasure. 3.3 details the hardening algorithm. 3.4 highlights the subtleties of implementing the hardening algorithm in LLVM. Finally, 3.5 discusses the design choices in a more general context.

As always, one can attempt to attack the countermeasure itself. The security theorem in Section 4 proves that no attempt at skipping, repeating or forging `ccs` instructions or jumps can succeed, leading to a strong security guarantee.

3.2 ISA and hardware extensions

We support the countermeasure with a custom ISA extension named Xccs for *Code CheckSum*[5]. Throughout this section, all examples refer to Figure 6, and $\mathtt{u}_N$ denotes the type of `N`-bit unsigned integers.

Xccs introduces four new Control and Status Registers (CSR), which together form the σ field of the program state:

- `CCS` : $\mathtt{u}_{32}$ is the running checksum for the current block. For example, the region of `g` from address `40108` through `40114` adds up in little-endian to `0x40b20505 + 0x00010141 + 0x0000100b = 0x40b31651` so the value of `CCS` at `40114` will be `0x40b31651` if no attack is performed.
- `CCSPROT` : $\mathtt{u}_{32}$ indicates the PC value at which a protected instruction (jump) is expected to execute. It is zero most of the time and non-zero for a single step after a checksum is validated. In Figure 6, it is set at `400f4` and `40118`.
- `CCSD` : $\{\mathtt{E} : \mathtt{u}_1, \mathtt{JO} : \mathtt{u}_5\}$ holds control information; the *enable* bit (`E`) indicates whether checksum protection is active for the current block (which currently is whenever PC $\geq$ `0x40000`), and the *jump offset* field (`JO`) is set by `ccscall` as described below.
- `CCSDS` : $\mathtt{u}_{32}$ ("delay slot") is used to implement the decoding of Xccs instructions' 32-bit checksum arguments. We leave discussion of this detail to supplementary material.

The meat of Xccs is the addition of four guard instructions whose encoding is shown in Figure 7, all of which are followed by a 32-bit ⟨**checksum**⟩ argument:

- **`ccs`** ⟨**checksum**⟩ compares the `CCS` register with the provided argument and traps if they differ. Otherwise, it sets `CCSPROT` = PC + 8 so a jump or function call can execute at the next step. For instance, at `40114` in Figure 6, the dynamic value of the `CCS` register is compared to `0x40b31651`. When no faults are injected, these are equal, so execution proceeds to the `c.ret` instruction.
- **`ccscall`** N ⟨**checksum**⟩ is similar; it is used before function calls. It sets `CCSD.JO` = N, which causes the next function call's return address to increase by $2N$ bytes. For instance, the `ccscallb 8` at `400ec` changes the return address after the `jalr` (call) instruction from `400f8` to `40108`, skipping over the `c.ebreak` block.
- **`ccsb`** and **`ccscallb`** are variations that flip the least significant bit (LSB) of the checksum before comparing it. This is leveraged to ensure that no ⟨**checksum**⟩ argument decodes as a jump or Xccs instruction, as detailed in Section 3.3. Again with Figure 6, the sum of the first block up to `400f0` is `0xc60675e3`, but that decodes as a `bltu`, creating a vulnerability. The LSB is flipped to avoid this.

[5]The "X" is standard notation for unofficial RISC-V extensions.

31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	
0	0	0	001	0	XCCS	ccs
0	0	0	010	N	XCCS	ccscall N
0	0	0	101	0	XCCS	ccsb
0	0	0	110	N	XCCS	ccscallb N

Figure 7. Xccs instructions (32-bit). `XCCS` is `0001011`.

All Xccs instructions further trap when run at unaligned PC, which prevents most attempts at forging them. Finally, existing CPU behavior is modified as follows:

1. All branch instructions trap if PC ≠ `CCSPROT`. If a branch occurs, `CCS` and `CCSPROT` are reset to 0.
2. Every other instruction traps if `CCSPROT` ≠ 0.
3. Every value retrieved from a fetch is added to `CCS`.
4. Call instructions add 2 × `CCSD.JO` to the return address and clear `CCSD.JO` before jumping.

Hardware support in this countermeasure serves an important dual purpose: it enforces the jail, and it updates the checksum through a system that is not vulnerable to attacks, so that checks of the trusted `CCS` value can be guaranteed even though the reference ⟨**checksum**⟩ is itself vulnerable.

3.3 Hardening algorithm

Algorithm 1 shows the hardening process for a single block in pseudocode, with our canonical example in Figure 6. (As all blocks are independent, this algorithm is executed by the compiler for every block in the program.)

The main **for** loop (line 15) iterates over the instructions of the original block. All instructions are copied to the hardened block (line 26). Jump instructions are preceded by a guard, which is `ccscall`/`ccscallb` (line 22) for function calls and `ccs`/`ccsb` for other jumps (line 24). `nop`s are used to ensure that guards are aligned and jumps are always separated by at least one non-jump instruction, both of which prevent subtle attacks against the countermeasure.

The second **for** loop (line 27) adds a barrier of `c.ebreak` instructions, which raise a distinctive exception when executed. Their role is to prevent control from leaving the block by skipping over the terminator. Up to $2N + 4$ are needed to address the worst case where control reaches the middle of a checksum whose second half is a 32-bit opcode, in which an attack of a $\mathtt{S\&R}_{32}$ followed by a $\mathtt{S}_{32(N)}$ would reach $4N + 6$ bytes past the checksum.

Procedure `addToBlock` appends instructions to the hardened block while computing the reference checksum value `sum` by summing instruction's opcodes. This accounts for instructions' alignment with the function

$$\mathtt{realign}(\mathtt{offset}, \mathtt{i}) = \begin{cases} \mathtt{i} \text{ if } \mathtt{offset} \equiv 0 \ [4] \\ 2^{16}\mathtt{LSH}(\mathtt{i}) + \mathtt{MSH}(\mathtt{i}) \text{ otherwise} \end{cases}$$

Calling convention: `a0` *serves as both argument and return value for* `f` *and* `g`. `ra` *is used for function calls and is caller-saved (push/pop omitted).*

```
1  g:  (...) # push ra to stack
2      PseudoCALL @f
3      # ^a0 forwarded implicitly
4
5
6      $a0 = nsw ADDI $a0, 1
7      (...) # pop ra from stack
8
9      PseudoRET $a0
10
```

(a) Machine IR before back-end pass.

```
1  g:  (...) # push ra
2      PseudoCALL @f, .LBB1_0
3      PseudoCCSTRAP 2
4
5  .LCCS_Region_Start0:
6      $a0 = nsw ADDI $a0, 1
7      (...) # pop ra
8      CCS .LCCS_Region_Start0
9      PseudoRET $a0
10     PseudoCCSTRAP 2
```

(b) Machine IR after back-end pass.

```
g: e4: 41 11 06 c6    (...)     # push ra
   e8: 97 00 00 00    auipc     ra, 0
   ec: 0b 24 00 00    ccscall   8
   # ▾ R_RISCV_CHECKSUM: g
   f0: 00 00 00 00    .word     0x00000000
   f4: e7 80 00 00    jalr      ra, 0(ra)
   f8: 02 90          c.ebreak  # 8 times (2N+4)
.LCCS_Region_Start0:
  108: 05 05          c.addi    a0, a0, 1
  10a: b2 40 41 01    (...)     # pop ra
  10e: 01 00          c.nop     # ccs alignment
  110: 0b 10 00 00    ccs
  # ▾ R_RISCV_CHECKSUM: .LCCS_Region_Start0
  114: 00 00 00 00    .word     0x00000000
  118: 82 80          c.ret
  11a: 02 90          c.ebreak  # 8 times
```

(c) Object code before linking.

Figure 8. Stages of hardening the program from Figures 1 and 6 ($N = 2$) in our LLVM implementation.

where $\mathsf{LSH}, \mathsf{MSH} : u_{32} \to u_{16}$ are the Most and Least Significant Halves respectively. This process is equivalent to summing the lines of the final layout table. Finally, `addChecksum` selects whether to use `ccs`/`ccscall` or their `-b` variants. The `-b` variants are selected when the checksum value is an "invalid checksum literal", i.e. it decodes as a jump, Xccs instruction or `c.ebreak`. Such values could be misused as instructions if an attacker were to skip the guard that precedes it. Flipping the LSB ensures that these sensitive values do not appear in code. Our running example in Figure 6 is obtained by executing this algorithm on both blocks of `g`'s original code from Figure 1 and linking it.

3.4 LLVM implementation

Algorithm 1 cannot be implemented as-is in a single pass in a standard compiler, because reference checksum values depend on the exact bit-level encoding of each instruction, which is not decided until the linker relocates references to globals and functions. See for instance how the call in Figure 8c (before linking) has placeholder zero-offsets but the one in Figure 6 (after linking) has a proper target offset.

We see this as a benign occurrence of a fundamental issue: the progressive lowering that standard toolchains go through is designed around *functional* invariants, not *security* invariants. Encodings are decided late because they don't matter to the compiler, which is only interested in the functional specification of instructions. The addition of a security countermeasure to the compiler breaks this assumption and creates a new concern in *threading the security transform through the abstraction gap of a functional lowering.*

In this case, we are able to implement the algorithm in two steps: a late Machine IR[6] pass followed by an extension to the linker relocation process.

- **Machine IR pass**: the program's Machine IR representation is first transformed late in the back-end (from Figure 8a to Figure 8b). This pass handles all tasks that *add* code into the program, including:
  - Aligning functions and blocks to 4-byte boundaries;
  - Adding aligned Xccs instructions before all Machine IR instructions that expand into RISC-V jumps, such as `PseudoRET`. A label indicates the start of the region that the checksum must cover;
  - Adding the trap barrier with the `PseudoCCSTRAP` N pseudo-instruction, which later expands into a series of $2N + 4$ `c.ebreak` instructions.

  At this stage some jumps are still hidden in pseudo-instructions, like the function call in `PseudoCALL`. This is because far jumps in RISC-V are implemented with a pair of instructions, `auipc` and a jump, to overcome the limited distance that can be encoded into single jump instructions. In LLVM, this is expanded later in the code emitter due to limitations in the back-end structure; we count this towards the Machine IR pass for simplicity of exposition. The Machine IR pass is followed by static branch relaxation (the unfolding of far jumps into multi-instruction sequences and converse compaction of near jumps into shorter instructions), which is the main reason why the addition of new code cannot be delayed more; it would break short jumps.[7]

[6] The back-end intermediate representation used by LLVM, which is aware of the target architecture, not the usual LLVM IR.

[7] We disable linker relaxation for simplicity to maintain jumps' alignment, but it could be enabled after adding appropriate alignment relocations.

Algorithm 1 Algorithm: HARDEN

Input: A source block $[\mathtt{i}_1, ..., \mathtt{i}_n]$
Input: Upper bound N for $\mathsf{S32}(k)$ rule ($k \leq N$)
Output: A hardened block hb.

```
1:  hb ← [ ]
2:  sum : u32 ← 0
3:  offset ← 0                                  ▷ Blocks are 4-aligned.
4:  procedure addToBlock(i)
5:      hb.append(i)
6:      sum ← sum + realign(offset, u32(i))
7:      offset ← offset + ‖i‖
8:  procedure addChecksum(i, ib)
9:      if sum + i is not a valid checksum literal then
10:         addToBlock(ib)
11:         addToBlock(sum ⊕ 1)
12:     else
13:         addToBlock(i)
14:         addToBlock(sum)
15: for i in [i1, ..., in] do
16:     if i is a jump instruction then
17:         if offset = 0 then                  ▷ No empty sections.
18:             addToBlock(encode(nop))
19:         else if offset ≡ 2 [4] then         ▷ Force alignment.
20:             addToBlock(encode(c.nop))
21:         if i is a function call then
22:             addChecksum(encode(ccscall (2N + 4)),
                            encode(ccscallb (2N + 4)))
23:         else                                ▷ Jumps/branches.
24:             addChecksum(encode(ccs), encode(ccsb))
25:         offset ← 0
26:     addToBlock(i)
27: for j = 1 to 2N + 4 do                      ▷ Add trap barrier.
28:     addToBlock(encode(c.ebreak))
29: return hb
```

During object file generation, the 8-byte CCS Machine IR instruction is replaced with a 4-byte Xccs opcode and a placeholder zero-checksum. A custom relocation of type R_RISCV_CHECKSUM (marked by a comment in Figure 8c) is added to mark the checksum region for the linker.[8]

- **Linking**: the linker follows relocation entries to compute checksums and insert them in the provided spaces. The linker script is also updated so that hardened objects are linked to a different virtual address (0x40000) than non-hardened libraries and runtime files (0x10000), which we use as the basis to enable Xccs protection.

3.5 Discussion

We discuss some of our design choices and how the scheme interacts with other micro-architectural features.

[8]The R_RISCV_CALL relocation for auipc/jump pairs is also replaced with a custom type to inform the linker of the newly-added ccs in the pair.

Fault's effect on CCS updates. Our countermeasure relies on CCS updates not being vulnerable to attacks. This is a reasonable inference based on the fault model: recall that fetch skips are induced by clock glitches, which are known to cause problems locally along critical signal propagation paths. Alshaer et al. [2022] identify that such paths are mostly in the fetch stage of the pipeline, but CCS can be updated in the execution stage using the realign sum technique, and is thus unlikely to be affected.

Interrupts. Common interrupts and signal handlers (that are invisible from the main thread) would not interfere with Xccs protections (with the only OS support needed being to save Xccs registers, which can be viewed as an extension of PC, to the CPU context structure). However, a non-returning interrupt (such as a signal exiting) would leave the current block without a check. We assume such a no-returning action implies abandoning the critical section where the interrupt occurred; otherwise, there might be a vulnerability.

Effect of faults on complex architectures. The fault model from [Alshaer et al. 2022] does not describe hardware responses to fault attacks during speculative or out-of-order execution. The study and design of fault models at the micro-architectural level is already state-of-the-art, and applying it to these complex features is a completely open problem. While Xccs is amenable to speculative execution (mispredictions would not lead to false checksum exceptions because the checksum resets at the beginning of every block) and out-of-order execution (the checksum update is associative-commutative, allowing for reordering within each block) it remains unlikely that clock glitches would affect such complex designs in the same way as the simple processors from which fetch skips are derived.

Possibility of a hardware-only solution. Hardware-only countermeasures against fault attacks present their own challenges [Clercq and Verbauwhede 2017]. Detecting the clock glitch at the (hardware) source using a *fault detector* [Gomina et al. 2014] creates a performance trade-off between sensitivity and the rate of false alarms, which limits the approach to security-critical systems that can afford the performance loss. Detecting corruption in the instruction stream requires extra hardware logic that risks being itself faulted (increased attack surface). Software/hardware propositions like our Xccs countermeasure have contrasting benefits. We minimize exposure to the fault because the detection only relies on CCS updates (which occur in the execution stage, away from disrupted fetch logic) and a checksum check made after the fault's transient effect has subsided. (We carefully discussed the safety of these operations with authors of [Alshaer et al. 2022].) We also get the benefits of using more software and less hardware, such as incurring costs only in critical sections and the possibility of implementing hardware support late or during minor hardware revisions.

Specificity of the attack model. Many (mostly early) works in fault literature attempt to protect against *all* program misbehaviors, described as "soft errors". By contrast, we target a single vulnerability, which might appear overly specific. However, the fetch skips model coined by [Alshaer et al. 2022] results from extensive physical injection campaigns, where it described the impact of 80-90% of clock glitches on Cortex-M boards [Alshaer 2023], making this countermeasure useful against common attack vectors on real boards. In addition, we argue that the lack of a precise definition for soft errors leads to tricky vulnerabilities[9] preventing any proof-based security standard from being met. This is why we focus on fetch skip attacks, for which we can formally prove security.

4 Security theorem

We now state the security theorem for our countermeasure and the main steps of the proof. We omit some invariants about σ/ρ whose significance is only clear with the full formalization and proof (see supplementary material).

We define a *step* as a statement

$$s = \langle \mathrm{PC}, \rho, \sigma, \alpha \rangle \rightarrow \langle \mathrm{PC}', \rho', \sigma', \alpha' \rangle$$

materializing the change in execution state associated with the execution of a single instruction, and an *execution* as a sequence of steps $[s_0, \ldots, s_n]$ such that the initial state of any s_i ($i > 0$) is the final state of s_{i-1}.

Theorem (simplified).
Let $P = [\mathsf{HARDEN}(\mathsf{bb}_1), \ldots \mathsf{HARDEN}(\mathsf{bb}_{|P|})]$ *a program hardened by Algorithm 1, and* $e = [s_0, \ldots, s_{|e|}]$ *an execution such that*

- s_0 *starts at the top of a block of* P*;*
- $s_{|e|}$ *successfully executes a program exit instruction.*

Then e *can be partitioned into sub-sequences* $(s_{t_i} \ldots s_{b_i})_{1 \le i \le m}$ *each executing one hardened block* $\mathsf{HARDEN}(\mathsf{bb}_i)$*, such that*

1. *Each* s_{t_i} *starts at the top of* $\mathsf{HARDEN}(\mathsf{bb}_i)$*;*
2. *Each* s_{b_i} $(i \neq m)$ *runs a legitimate (non-forged) jump instruction of* bb_i*, and* $\sigma_{b_i}.\mathsf{CCS}$ *is the correct associated checksum;*
3. *If each segment* $s_{t_i} \ldots s_{b_i}$ *contains at most one faulted fetch and the last segment* $s_{t_m} \ldots s_{b_m}$ *has none*[10]*, then the entire execution* e *contains no faults.*

In essence, this theorem says that the countermeasure guarantees the execution of the original sequence of instructions (as witnessed by the checksum) at every control flow edge. The key property is the *local security* at the block level; the security property of passing checksums is checked at every block, and deviations from that property are detected before the block ends. This facilitates both formal analysis (by removing difficulties associated with control flow) and testing (by providing an easy way to detect failures in the countermeasure).

The progression of the argument is as follows:

1. Control cannot reach the end of a protected block due to the `c.ebreak`-based trap barrier;
2. Control can only leave a protected block through a jump if it passes the associated checksum;
3. Using exactly one faulted fetch in the execution of a block always invalidates the checksum.

The checksum method by itself does not always guarantee that the execution cannot be compromised, as a vulnerable block could contain a *checksum collision*: an unintended execution path, reachable with multiple faults, whose checksum is the same as the intended execution. Such collisions can be detected statically after linking, but they are hard to solve because the linker is unable to add or remove instructions to prevent the collision once it is discovered, as the first could break short jumps and both would change jump offsets thus more checksums. In this case, the toolchain's abstraction stack that we worked around fairly easily in Section 3.4 poses a greater challenge.

Note that the countermeasure still guarantees that checksums must pass even in multi-attack scenarios. Collisions are rare (as showcased in Section 5) despite the apparent weakness of the summing algorithm, mainly because the attacker cannot feed arbitrary inputs to the sum; they can only skip or repeat predetermined inputs. Thus, the scheme retains value as a practical protection mechanism against multi-fault attacks.

5 Implementation setting and evaluation

We now describe the implementation and the experimental settings we used to evaluate the countermeasure. We examine functional correctness, security guarantees, and cost.

5.1 Implementation and experimental setting

We implemented the hardening scheme in LLVM [Lattner and Adve 2004] 12.0,[11] linking programs with an off-the-shelf GNU C library using a modified GNU ld 2.40 linker. The hardware extension Xccs was prototyped in the system emulator QEMU 8.0 [Bellard 2005] and the processor simulator Gem5 22.1 [Lowe-Power et al. 2020]. The implementation is about 1150 lines in LLVM/ld for the countermeasure, and 850 lines in QEMU/Gem5 for evaluation.[12]

We evaluate our countermeasure on programs from the MiBench benchmark suite [Guthaus et al. 2001], which is

[9] For instance, triplication countermeasures such as SWIFT-R [Chang et al. 2006] and NEMESIS [Didehban et al. 2017] tend to assume that a single "soft error" only affects one of the three execution streams, but this is not true of e.g. the kind of decoding errors mentioned in [Didehban et al. 2017].
[10] We do not protect the exit of the last block because in practice the last instruction is a syscall invocation in the non-hardened libc function `exit()`.

[11] All the modified tools used in this paper are publicly available at gricad-gitlab.univ-grenoble-alpes.fr/michelse/fetch-skips-hardening.
[12] Gem5 currently only emulates 64-bit RISC-V instructions; we modified it to support 32-bit instructions for this evaluation.

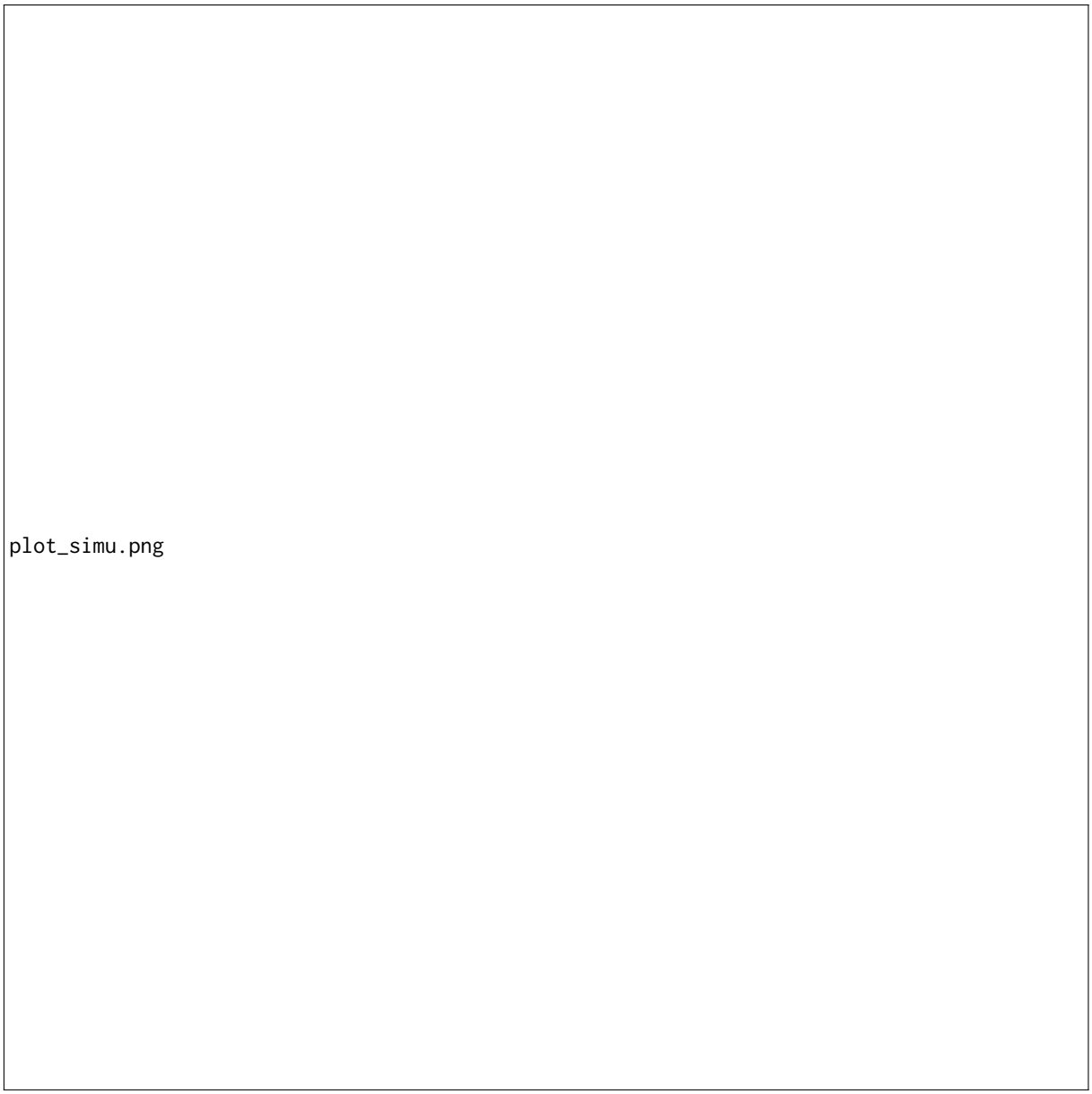

Figure 9. Outcomes of exhaustive S32(1), S32(2) (bars 1 and 2) and S&R32 (bars 3) single-fault injections, and random multi-fault injections on MiBench programs (bars R). The total number of injected faults is given for each program.

designed to be representative of embedded applications. We do not specifically use a cryptography- or security-oriented benchmark as we protect code at a low enough level that the nature of the program is not significant. Programs were hardened with $N = 2$, i.e. assuming a strong attacker that can inject double-skips every fetch. We perform three types of experiments, on a standard x86_64 GNU/Linux machine:

- Exhaustive single-fault injection campaigns in QEMU;
- Random multi-fault injection campaigns in QEMU;
- No-fault performance simulations in Gem5.

We instrument QEMU to support fault injections by skipping or replacing data during the transpilation step. We take advantage of the locally-correct design of the countermeasure and raise an explicit "countermeasure bypassed" exception at every successful exit of a block where a fault was triggered. This ensures that successful fault attacks are reported and cannot be masked by program logic. We similarly extend Gem5 with the decoding and execution of Xccs instructions, but leave out exceptional conditions since we use it only to evaluate performance without fault injections.

5.2 Functional correctness

We first assert that hardening preserves the functional behavior of programs when no fault is injected. This is checked

by running hardened MiBench programs and comparing their output to a non-cross-compiled x86 build and a non-hardened RISC-V build.[13] All programs pass this test.

5.3 Security guarantee

We then subject each program to two injection settings, both performed by emulation in QEMU:

1. *Single-fault exhaustive injection*: we attack every PC in the protected text segment of the program with single-fault S32(1), S32(2), and S&R32.
2. *Multi-fault random injection*: we attack 2000 code intervals of 64 bytes with randomly-selected sequences of 2 to 6 faults in close succession.

This showcases the guarantees proven in Section 4. Outcomes of injections for which the targeted PC was reached are classified in four categories in Figure 9:

- **Fault reported**: An invalid checksum, illegal jump, or other Xccs-imposed constraint was violated; or `c.ebreak` was executed.
- **Segfault**: The program segfaulted from an incorrect memory access as a result of the injected fault.
- **Other crash**: Illegal instructions being decoded by the CPU (`SIGILL`) or other rare crashes such as `SIGBUS`.
- **Countermeasure bypassed**: The program successfully exited a block after attacking it. This did not happen in any of the tests.

Figure 9 shows that no attack was able to bypass the protection from the countermeasure, which in the multi-fault case means that no checksum collision occurred.

A significant majority of faults that are reached result in Xccs violations, meaning that failing checksums and invalid attempts to jump out of blocks are adequately reported. Because all exits of attacked blocks were guarded with "Countermeasure bypassed" exceptions, which were not reached, every crash we encountered also occurred within the attacked block. This shows that the countermeasure fulfills its goal of containing faults within blocks.

Interestingly, crashes alone do not provide sufficient security: unprotected programs (not depicted) experience ∼90% crashes but only ∼30% in the block where the fault is injected. So like with instruction skip, a system facing random failures could leave the code unprotected, but the threat of targeted attacks requires a countermeasure like Xccs.

5.4 Performance

On average, hardening with $N = 2$ increased the size of protected functions (i.e., excluding libraries and runtime files) by a factor of 2.46 (which is large but usual for security applications especially against skips, due to duplication). Individual differences are given in Figure 10; unsurprisingly, programs with longer straight-line sections see less of an increase, while short, loop-intensive programs like `dijkstra` and `bitcount` get the largest overhead.

We run Gem5 simulations for each program[14] to estimate the overhead in execution time, also reported in Figure 10. The simulated system mainly consists of a 1-GHz RISC-V core with a 1600 MHz DDR3 controller simulated by Gem5's *timing* model.

The main cost is the execution of Xccs instructions themselves, which incur extra fetch-decode-execute cycles. As expected, programs with denser control flow in hot sections such as `dijkstra` and `bitcount` see by far the largest difference. We expected the code size increase to reduce cache efficiency, and ran simulations with an 8-kB 4-way instruction cache (the median size of hardened sections in our test, which is a lot smaller than the total text segment). This speeds up executions significantly (not pictured) but the ratio of hardened to non-hardened speed remains consistent. The overall performance hit is 10.2%, consistent with the knowledge that MiBench programs average out 6–7 instructions per basic block (keeping in mind that our blocks are longer than traditional basic blocks). Unsurprisingly, cryptographic schemes have the best performance results due to being computation-heavy with long blocks.

Existing countermeasures without hardware support have a comparatively much larger overhead. Geier et al. [2023] compare multiple combinations of countermeasures against single- to quadruple-instruction skips on a secure boot program. Only 3 combinations (nZDC + RACFED, NEMESIS + RACFED, and CompaSec) close more than 75% of vulnerabilities, all with space overhead above x4.85 and time overhead above x5.01.

6 Related work

The contribution of this paper follows the now numerous developments in the area of compilation for security. Recent countermeasure designs have consistently involved compilation tools [Barry, Couroussé, and Robisson 2016; Proy et al. 2017; Winderix et al. 2021], and issues stemming from cross-layer abstractions have been raised previously [Barry, Couroussé, Robisson, and Heydemann 2017]. Our work differs in its deliberate involvement of hardware, whereas existing designs that account for non-ISA details (such as timings in [Winderix et al. 2021]) still end up performing pure assembly transformations.

Countermeasure designs that lay more heavily towards hardware also exist. Manssour et al. [2022] show how hardware support can improve a classic countermeasure against data corruption which consists in executing instructions multiple times and comparing results. This is traditionally very costly due not only to repeated execution, but also program size, register pressure, and the addition of many branches

[13] For some programs, comparison with the native build is skipped due to differences in the precision of the math library.

[14] Except `patricia`, due to a 32- vs. 64-bit compatibility issue related to reused opcodes in the RISC-V ISA.

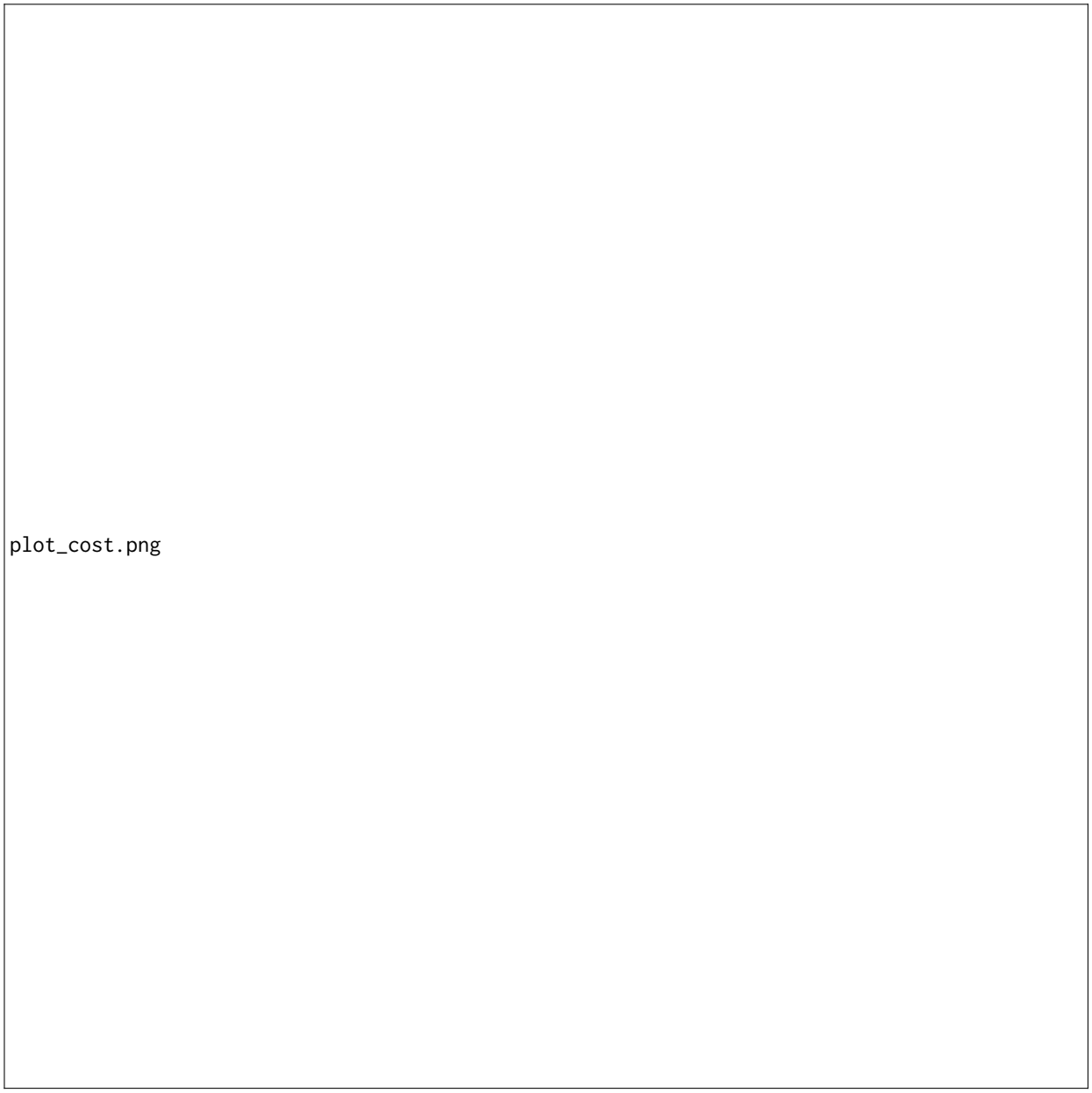

Figure 10. Evaluation of the cost of our counter-measure: static and dynamic metrics.

for comparisons. In this work, the ISA is extended with a new "replay" instruction `rpl w n` which repeats the next w instructions n times, and compares in hardware with an exception in case of a mismatch. A benefit is that the compiler only needs to mark sensitive instructions, however this shifts more security burden to hardware and increases the effort for hardware qualification/certification compared to a more lightweight extension like Xccs.

Our countermeasure is based on classical redundancy ideas and draws inspiration from previous work on instruction skips [Yao and Schaumont 2018] and control-flow integrity protections [Mishra et al. 2022; Zgheib et al. 2022]. Complications associated with multi-fault attacks incentivized us to aim for a design that guarantees security *locally* (at every block) to keep formal reasoning simple. Many countermeasures in security literature (for instance [Chang et al. 2006; Didehban et al. 2017; Geier et al. 2023]) achieve sophisticated and carefully-balanced compiler transforms, but lack a detailed enough programming and fault model to allow for a formal proof of the security property. We believe that our semantics approach that includes "just enough" details in the programming model, encodes the fault model in the semantics, and encourages formal proofs, could be generalized for other low-level countermeasures.

As for the correctness of securization schemes, apart from manual proofs (in which this work falls), automatic countermeasure verification techniques includes translation validation [Busi et al. 2022], static analysis [Christofi et al. 2013], and symbolic execution [Potet et al. 2014]. These usually focus on proving properties of the form "either the attack is detected or the program crashed", as we did here. Another remarkable piece of work in this area is a proof that the C compiler CompCert [Leroy 2009] preserves the constant-time countermeasure when applied to source code [Hutin 2021]. A possible extension of our work would be to further validate through one of these approaches.

7 Conclusion

We have presented a new countermeasure to a micro-architectural fault attack targeting the instruction fetch unit in RISC-V processors. Our approach combines a hardware extension of the RISC-V ISA, dubbed Xccs, and a machine code transformation performed at compile- and link-time. The design process was guided by a specially-tailored semantics that captures just the right set of architectural details to formalize the hardening process and enable a proof of security. We evaluated the scheme's security by emulating both exhaustive single-fault and random multi-fault injection campaigns, and its performance with a processor simulation. To our knowledge, the present contribution is the first example of such a proven countermeasure for a micro-architectural fault model; which notably connects recent security experts' results to compiler back-end techniques.

We believe that this approach is applicable more generally to the design of code hardening schemes that guarantee both security and safety. We implemented this particular countermeasure in the late back-end, which is pretty standard for low-level attacks, and keeps the abstraction gap between the attack and countermeasure fairly small. In future work we wish to explore countermeasures implemented at other (sometime multiple) levels in the compilation flow, widening that gap. This will escalate toolchain integration challenges already raised in this paper, as the compiler's heavy descent in abstraction might break or hinder security properties. Partial solutions such as [Vu 2021], which advocates for embedding security properties into functional properties, leads us to believe that cleaner, more methodical integration is possible. Similar improvements to the semantic model appear needed to mirror this property preservation in the security proof.

Data-availability statement

An artifact of this work is available for reproduction on Zenodo [Michelland 2024] and includes pre-built tools and reference outputs. Source code and documentation for the tools (LLVM, GNU binutils, QEMU, Gem5) is further available for reuse through the Git repository at gricad-gitlab.univ-grenoble-alpes.fr/michelse/fetch-skips-hardening.

References

Ihab Alshaer. Oct. 2023. "Cross-Layer Fault Analysis for Microprocessor Architectures". PhD thesis. Université Grenoble Alpes [2020-....], (Oct. 2023). https://www.theses.fr/s247831.

Ihab Alshaer, Brice Colombier, Christophe Deleuze, Vincent Beroulle, and Paolo Maistri. 2022. "Variable-Length Instruction Set: Feature or Bug?" In: Maspalomas, Spain. IEEE. ISBN: 978-1-6654-7405-4. DOI: 10.1109/DSD57027.2022.00068.

H. Bar-El, Hamid Choukri, D. Naccache, Michael Tunstall, and C. Whelan. 2006. "The Sorcerer's Apprentice Guide to Fault Attacks". *Proceedings of the IEEE*, 94, 2, 370–382. DOI: 10.1109/JPROC.2005.862424.

Thierno Barry, Damien Couroussé, and Bruno Robisson. Jan. 2016. "Compilation of a Countermeasure Against Instruction-Skip Fault Attacks". In: *Workshop on Cryptography and Security in Computing Systems* (Proceedings of the Third Workshop on Cryptography and Security in Computing Systems). vienna, Austria, (Jan. 2016). DOI: 10.1145/2858930.2858931.

Thierno Barry, Damien Couroussé, Bruno Robisson, and Karine Heydemann. Mar. 2017. "Automated Combination of Tolerance and Control Flow Integrity Countermeasures against Multiple Fault Attacks". In: *European LLVM Developers Meeting*. Saarbrücken, Germany, (Mar. 2017). https://hal.sorbonne-universite.fr/hal-01660160.

Gilles Barthe, François Dupressoir, Pierre-Alain Fouque, Benjamin Grégoire, and Jean-Christophe Zapalowicz. 2014. "Synthesis of Fault Attacks on Cryptographic Implementations". In: *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security* (CCS '14). Association for Computing Machinery, Scottsdale, Arizona, USA, 1016–1027. ISBN: 9781450329576. DOI: 10.1145/2660267.2660304.

Fabrice Bellard. 2005. "QEMU, a Fast and Portable Dynamic Translator". In: *Proceedings of the Annual Conference on USENIX Annual Technical Conference* (ATEC '05). USENIX Association, Anaheim, CA, 41.

Matteo Busi, Pierpaolo Degano, and Letterio Galletta. 2022. "Towards Effective Preservation of Robust Safety Properties". In: *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing* (SAC '22). Association for Computing Machinery, Virtual Event, 1674–1683. ISBN: 9781450387132. DOI: 10.1145/3477314.3507084.

J. Chang, G.A. Reis, and D.I. August. 2006. "Automatic Instruction-Level Software-Only Recovery". In: Philadelphia, PA, USA. IEEE, Philadelphia, PA, USA, 83–92. ISBN: 0-7695-2607-1. DOI: 10.1109/DSN.2006.15.

Maria Christofi, Boutheina Chetali, Louis Goubin, and David Vigilant. 2013. "Formal Verification of a CRT-RSA Implementation Against Fault Attacks". *Journal of Cryptographic Engineering*, 3, 3, 157–167. DOI: 10.1007/s13389-013-0049-3.

Ruan de Clercq and Ingrid Verbauwhede. 2017. "A survey of Hardware-based Control Flow Integrity (CFI)". *CoRR*, abs/1706.07257. http://arxiv.org/abs/1706.07257 arXiv: 1706.07257.

Moslem Didehban, Aviral Shrivastava, and Sai Ram Dheeraj Lokam. Nov. 2017. "NEMESIS: a software approach for computing in presence of soft errors". In: *Proceedings of the 36th International Conference on Computer-Aided Design* (ICCAD '17). IEEE Press, Irvine, California, (Nov. 2017), 297–304.

Johannes Geier, Lukas Auer, Daniel Mueller-Gritschneder, Uzair Sharif, and Ulf Schlichtmann. Jan. 2023. "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). Association for Computing Machinery, Tokyo, Japan, (Jan. 2023), 676–682. ISBN: 9781450397834. DOI: 10.1145/3566097.3567925.

Kamil Gomina, Jean-Baptiste Rigaud, Philippe Gendrier, Philippe Candelier, and Assia Tria. 2014. "Power supply glitch attacks: Design and evaluation

of detection circuits". In: *2014 IEEE International Symposium on Hardware-Oriented Security and Trust (HOST)*, 136–141. DOI: 10.1109/HST.2014.6855584.

Matthew R Guthaus, Jeffrey S Ringenberg, Dan Ernst, Todd M Austin, Trevor Mudge, and Richard B Brown. 2001. "MiBench: A free, commercially representative embedded benchmark suite". In: *Proceedings of the fourth annual IEEE international workshop on workload characterization. WWC-4 (Cat. No. 01EX538)*. IEEE, 3–14.

Andrea Höller, Armin Krieg, Tobias Rauter, Johannes Iber, and Christian Kreiner. 2015. "QEMU-Based Fault Injection for a System-Level Analysis of Software Countermeasures Against Fault Attacks". In: *2015 Euromicro Conference on Digital System Design*, 530–533. DOI: 10.1109/DSD.2015.79.

Rémi Hutin. 2021. "Verified Secure Compilation against Timing Side-Channels". PhD thesis. http://www.theses.fr/2021ENSR0029/document. 2021ENSR0029.

Ronan Lashermes, Hélène Le Bouder, and Gaël Thomas. Nov. 2018. "Hardware-Assisted Program Execution Integrity: HAPEI". In: *NordSec 2018 : 23rd Nordic Conference on Secure IT Systems*. Oslo, Norway, (Nov. 2018). DOI: 10.1007/978-3-030-03638-6_25.

Chris Lattner and Vikram Adve. 2004. "LLVM: A compilation framework for lifelong program analysis & transformation". In: *International symposium on code generation and optimization, 2004. CGO 2004.* IEEE, 75–86.

Johan Laurent. Nov. 2020. "Modélisation de fautes utilisant la description RTL de microarchitectures pour l'analyse de vulnérabilité conjointe matérielle-logicielle". Theses. Université Grenoble Alpes, (Nov. 2020). https://tel.archives-ouvertes.fr/tel-03167493.

Johan Laurent, V. Beroulle, C. Deleuze, Florian Pebay-Peyroula, and Athanasios Papadimitriou. 2018. "On the Importance of Analysing Microarchitecture for Accurate Software Fault Models", 561–564. DOI: 10.1109/DSD.2018.00097.

Xavier Leroy. July 2009. "Formal Verification of a Realistic Compiler". *Commun. ACM*, 52, 7, (July 2009), 107–115. DOI: 10.1145/1538788.1538814.

Jason Lowe-Power et al.. 2020. "The gem5 simulator: Version 20.0+". *arXiv preprint arXiv:2007.03152*.

Noura Ait Manssour, Vianney Lapôtre, Guy Gogniat, and Arnaud Tisserand. 2022. "Processor Extensions for Hardware Instruction Replay against Fault Injection Attacks". In: *2022 25th International Symposium on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, 26–31. DOI: 10.1109/DDECS54261.2022.9770170.

[SW] Sébastien Michelland, *Replication package for article: From low-level fault modeling (of a pipeline attack) to a proven hardening scheme* Jan. 2024. DOI: 10.5281/zenodo.10440364, URL: https://doi.org/10.5281/zenodo.10440364.

Tanmaya Mishra, Thidapat Chantem, and Ryan Gerdes. Oct. 2022. "Survey of Control-Flow Integrity Techniques for Real-Time Embedded Systems". *ACM Trans. Embed. Comput. Syst.*, 21, 4, (Oct. 2022). DOI: 10.1145/3538275.

Marie-Laure Potet, Laurent Mounier, Maxime Puys, and Louis Dureuil. 2014. "Lazart: A Symbolic Approach for Evaluation the Robustness of Secured Codes against Control Flow Injections". In: *2014 IEEE Seventh International Conference on Software Testing, Verification and Validation*, 213–222. DOI: 10.1109/ICST.2014.34.

Julien Proy, Karine Heydemann, Alexandre Berzati, and Albert Cohen. Dec. 2017. "Compiler-Assisted Loop Hardening Against Fault Attacks". *ACM Trans. Archit. Code Optim.*, 14, 4, (Dec. 2017). DOI: 10.1145/3141234.

Carlton Shepherd, Konstantinos Markantonakis, Nico van Heijningen, Driss Aboulkassimi, Clément Gaine, Thibaut Heckmann, and David Naccache. Dec. 2021. "Physical fault injection and side-channel attacks on mobile devices: A comprehensive analysis". *Computers & Security*, 111, (Dec. 2021), 102471. DOI: 10.1016/j.cose.2021.102471.

The RISC-V Instruction Set Manual Vol. I. (Dec. 2019).

Stefan Tillich, Christoph Herbst, and Stefan Mangard. 2007. "Protecting AES Software Implementations on 32-Bit Processors Against Power Analysis". In: *Applied Cryptography and Network Security, 5th International Conference, ACNS 2007, Zhuhai, China, June 5-8, 2007, Proceedings* (Lecture Notes in Computer Science). Ed. by Jonathan Katz and Moti Yung. Vol. 4521. Springer, 141–157. DOI: 10.1007/978-3-540-72738-5_10.

Simon Tollec, Mihail Asavoae, Damien Couroussé, Karine Heydemann, and Mathieu Jan. 2022. "Exploration of Fault Effects on Formal RISC-V Microarchitecture Models". In: *2022 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, 73–83. DOI: 10.1109/FDTC57191.2022.00017.

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

Vincent Werner, Laurent Maingault, and Marie-Laure Potet. Oct. 31, 2022. "An end-to-end approach to identify and exploit multi-fault injection vulnerabilities on microcontrollers". *Journal of Cryptographic Engineering*, (Oct. 31, 2022), 1–17. DOI: 10.1007/s13389-022-00292-z.

Hans Winderix, J. Mühlberg, and F. Piessens. 2021. "Compiler-Assisted Hardening of Embedded Software Against Interrupt Latency Side-Channel Attacks", 667–682. DOI: 10.1109/EuroSP51992.2021.00050.

Yuan Yao and Patrick Schaumont. 2018. "A Low-Cost Function Call Protection Mechanism Against Instruction Skip Fault Attacks". In: *Proceedings of the 2018 Workshop on Attacks and Solutions in Hardware Security* (ASHES '18). Association for Computing Machinery, Toronto, Canada, 55–64. ISBN: 9781450359962. DOI: 10.1145/3266444.3266453.

Bilgiday Yuce, Nahid Farhady Ghalaty, Harika Santapuri, Chinmay Deshpande, Conor Patrick, and Patrick Schaumont. 2016. "Software Fault Resistance is Futile: Effective Single-Glitch Attacks". In: *2016 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, 47–58. DOI: 10.1109/FDTC.2016.21.

Anthony Zgheib, Olivier Potin, Jean-Baptiste Rigaud, and Jean-Max Dutertre. 2022. "A CFI Verification System based on the RISC-V Instruction Trace Encoder". In: *2022 25th Euromicro Conference on Digital System Design (DSD)*, 456–463. DOI: 10.1109/DSD57027.2022.00067.

Supplementary material for the CC'24 submission:
From low-level fault modeling (of a pipeline attack) to a proven hardening scheme

A Operational semantics

In this appendix, we formalize operational semantics for a minimal, representative subset of 32-bit RISC-V assembly which includes fetch skip attacks and our Xccs extension, and provide a detailed proof of the claims made in Section 4.

Notations. We write u_N the type of N-bit unsigned integers. We also use structure-like notations $\{\langle\text{field}\rangle : \langle\text{type}\rangle, \ldots\}$ and $\langle\text{struct}\rangle.\langle\text{field}\rangle$ for bit fields and named collections of values.

A.1 Program and execution models

Definition 1 (Instruction, Block, Program)**.**
*An **instruction** $\mathtt{i}$ is a 16- or 32-bit integer (a u_{16} or a u_{32}) corresponding to a RISC-V opcode.*[15] *We write $\|\mathtt{i}\|$ the size of an instruction in bytes (2 or 4).*

*A **block** $\mathtt{bb}$ is a nonempty sequence of instructions, along with a 4-aligned address value $\mathtt{blockAddr(bb)}$ indicating where it is loaded in memory.*

*A **program** P is a collection of non-intersecting blocks.*

Since instructions in the same block are loaded contiguously, each block spans the interval

$$\mathtt{blockSpan(bb)} = \left[\mathtt{blockAddr(bb)},\ \mathtt{blockAddr(bb)} + \sum_{\mathtt{i} \in \mathtt{bb}} \|\mathtt{i}\|\right).$$

Programs are assumed to be well-formed, in that no two blocks' spans intersect and every jump points to the $\mathtt{blockAddr}$ of some block.

Definition 2 (Program state)**.**
A program state in RISC-V assembler with Xccs and fetch skips is a quadruplet $\langle PC, \rho, \sigma, \alpha\rangle$, where

- $PC : \mathsf{u}_{32}$ *is the program counter;*
- $\rho : \mathsf{u}_{32}$ *is the last row fetched from code memory;*
- $\sigma = \{\mathtt{CCS}, \mathtt{CCSD}, \mathtt{CCSDS}, \mathtt{CCSPROT}\}$ *holds Xccs registers:*
  - $\mathtt{CCS} : \mathsf{u}_{32}$ *the running checksum of the current block;*
  - $\mathtt{CCSD} : \{\mathtt{E} : \mathsf{u}_1, \mathtt{J0} : \mathsf{u}_5\}$ *the Xccs status register;*
  - $\mathtt{CCSDS} : \mathsf{u}_{32}$ *the Xccs "delay slot" register;*
  - $\mathtt{CCSPROT} : \mathsf{u}_{32}$ *the protected PC register.*
- $\alpha : \mathsf{u}_{32}[32]$ *is the architectural state (registers* $\mathtt{x0}$*...*$\mathtt{x31}$*).*

The program state tracks the progress of execution. For simplicity, we do not include memory, which would be handled like the registers in α, or side-effects, which could be recorded as a trace whenever a side-effecting syscall is invoked by the $\mathtt{ecall}$ instruction. We only consider the final state of registers.

[15] There is no risk of type confusion because 16- and 32-bit instructions differ on their low bits.

Note that ρ is a raw 32-bit value which is not necessarily a legal instruction because instructions are not always 32-bit and not always aligned in memory.

Definition 3 (Step, Execution)**.**
*A **step** is a statement representing the effect of executing one instruction, written $\langle PC, \rho, \sigma, \alpha\rangle \to r$ where r is either a program state or one of two termination reasons:*

- $\bot$*, denoting an exception or crash;*
- $\mathsf{end}(\alpha)$*, denoting successful completion with final state α.*

*An **execution** is a sequence of program states ending with a termination reason, such that all pairs of consecutive elements are related by a step:*

$$\langle PC, \rho, \sigma, \alpha\rangle \to \ldots \to \langle PC_n, \rho_n, \sigma_n, \alpha_n\rangle \to \begin{cases} \bot \\ \mathsf{end}(\alpha') \end{cases}$$

A.2 Fetch and step rules

We can now describe how execution steps are derived from the fetch-decode-execute cycle of the CPU, including fault injections during fetches.

Definition 4 (Fetch)**.**
*A **fetch** is a statement $(PC, \rho)\ a \Rightarrow d\ (PC', \rho')$ representing the CPU fetching 32 bits at 4-aligned address $a : \mathsf{u}_{32}$ and getting value $d : \mathsf{u}_{32}$ ("data"), with the PC and ρ members of the program state being updated in the process.*

Each fault attack corresponds to one use of the $\mathsf{S32}(k)$ or $\mathsf{S\&R32}$ rules. Notice, for example, how rule $\mathsf{S32}(k)$ skips over a directly to $a + 4k$. In the following rules, $[a]$ denotes the 32 bits stored in memory at address a.

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

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

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

Figure 11. Fetch rules.

From there, we can formalize execution steps as an optional fetch, then the execution of a decoded instruction. This is shown in Figure 12. The $[\![\cdot]\!]$ and $[\![\cdot]\!]_{ccs}$ functions represent the semantics of individual instructions and are specified later. Both are oblivious to ρ so we re-insert ρ into their result after-the-fact using the $\bullet$ function. The functions $\mathtt{LSH}, \mathtt{MSH} : \mathsf{u}_{32} \to \mathsf{u}_{16}$ extract the least and most significant halves of a 32-bit value.

These inference rules for steps formalize the patterns shown in Figure 2 (ALIGNED-* is 2.(a) and (b), UNALIGNED-16 is

$$\textsc{Aligned-16}\quad \frac{\text{PC aligned} \qquad (\text{PC},\rho)\ \text{PC} \Rightarrow d\ (\text{PC}_F,\rho') \qquad \sigma.\texttt{CCSDS} = 0 \qquad \texttt{LSH}(d)\text{ is a 16-bit instruction}^{16}}{\langle \text{PC},\rho,\sigma,\alpha\rangle \longrightarrow [\![\texttt{LSH}(d)]\!](\text{PC}_F,\sigma,\alpha)\bullet\rho'}$$

$$\textsc{Aligned-32}\quad \frac{\text{PC aligned} \qquad (\text{PC},\rho)\ \text{PC} \Rightarrow d\ (\text{PC}_F,\rho') \qquad \sigma.\texttt{CCSDS} = 0 \qquad \texttt{LSH}(d)\text{ is a 32-bit instruction leader}^{16}}{\langle \text{PC},\rho,\sigma,\alpha\rangle \longrightarrow [\![d]\!](\text{PC}_F,\sigma,\alpha)\bullet\rho'}$$

$$\textsc{Unaligned-16}\quad \frac{\text{PC unaligned} \qquad \sigma.\texttt{CCSDS} = 0 \qquad \texttt{MSH}(\rho)\text{ is a 16-bit instruction}^{16}}{\langle \text{PC},\rho,\sigma,\alpha\rangle \longrightarrow [\![\texttt{MSH}(\rho)]\!](\text{PC},\sigma,\alpha)\bullet\rho'}$$

$$\textsc{Unaligned-32}\quad \frac{\text{PC unaligned} \qquad \texttt{MSH}(\rho)\text{ is a 32-bit instruction leader}^{16} \qquad \sigma.\texttt{CCSDS} = 0 \qquad (\text{PC},\rho)\ \text{PC}+2 \Rightarrow d\ (\text{PC}_F,\rho')}{\langle \text{PC},\rho,\sigma,\alpha\rangle \longrightarrow [\![2^{16}\texttt{MSH}(\rho)+\texttt{LSH}(d)]\!](\text{PC}_F,\sigma,\alpha)\bullet\rho'}$$

$$\textsc{Checksum-Delay-Slot}\quad \frac{\text{PC aligned} \qquad \sigma.\texttt{CCSDS} \neq 0 \qquad (\text{PC},\rho)\ \text{PC} \Rightarrow d\ (\text{PC}_F,\rho')}{\langle \text{PC},\rho,\sigma,\alpha\rangle \longrightarrow [\![\sigma.\texttt{CCSDS}]\!]_{ccs}(\text{PC}_F,\sigma,\alpha,d)\bullet\rho'}$$

Re-inserting ρ into the result of $[\![\cdot]\!]$ and $[\![\cdot]\!]_{ccs}$:

$$\langle \text{PC},\sigma,\alpha\rangle\bullet\rho = \langle \text{PC},\rho,\sigma,\alpha\rangle$$
$$\bot\bullet\rho = \bot$$
$$\texttt{end}(\alpha)\bullet\rho = \texttt{end}(\alpha)$$

Figure 12. Step rules.

2.(c) and Unaligned-32 is 2.(d)). For instance, in Unaligned-32, $\texttt{MSH}(\rho)$ identifies the two remaining bytes in the previously-fetched line as 32-bit instruction leader, so the rule fetches the next line d at PC + 2 and recombines ρ with $\texttt{LSH}(d)$ to form the complete 32-bit instruction and run it.

The extra rule Checksum-Delay-Slot describes the special treatment of the 32-bit checksum value following an Xccs instruction, which is triggered by $\sigma.\texttt{CCSDS} \neq 0$ and is described later with the semantics of Xccs instructions.

A.3 Instruction semantics

We now define instruction semantics through the functions

$$[\![\cdot]\!] : (\text{PC},\sigma,\alpha) \mapsto \langle \text{PC}',\sigma',\alpha'\rangle \text{ or } \bot \text{ or } \texttt{end}(\alpha')$$
$$[\![\cdot]\!]_{ccs} : (\text{PC},\sigma,\alpha,d) \mapsto \langle \text{PC}',\sigma',\alpha'\rangle \text{ or } \bot \text{ or } \texttt{end}(\alpha')$$

The two functions are similar in nature; $[\![\cdot]\!]_{ccs}$ is used when re-running Xccs instructions after their 32-bit checksum value (passed as 4th argument) has been found.

In the following, we assume a function encode which encodes assembler notation into 16/32-bit values (16-bit for mnemonics starting with "c.", 32-bit otherwise), following the RISC-V ISA [*The RISC-V Instruction Set Manual Vol. I* 2019]. Unspecified members in the output σ'/α' are implicitly kept from the input σ/α.

Instructions that are not jumps, traps, or Xccs instructions have their natural semantics, with two small changes: they cannot run when CCSPROT is set to a non-zero address, and they update CCS as a side-effect. We illustrate this category with the add and c.add instructions, and the c.nop instruction that we use as padding while hardening.

- $\texttt{i} = \texttt{encode}(\texttt{add}\ \texttt{r}_\texttt{d}, \texttt{rs}_1, \texttt{rs}_2)$, $\texttt{i} = \texttt{encode}(\texttt{c.add}\ \texttt{r}_\texttt{d}, \texttt{rs}_1, \texttt{rs}_2)$:

$$[\![\texttt{i}]\!](\text{PC},\sigma,\alpha) = \begin{cases} \bot \text{ if } \sigma.\texttt{CCSPROT} \neq 0 \\ \langle \text{PC} + \|\texttt{i}\|, \sigma', \alpha'\rangle \text{ otherwise, with} \\ \quad \sigma'.\texttt{CCS} = \sigma.\texttt{CCS} + \texttt{realign}(\text{PC}, \texttt{u}_{32}(\texttt{i})) \\ \quad \alpha'[\texttt{r}_\texttt{d}] = \alpha[\texttt{rs}_1] + \alpha[\texttt{rs}_2] \end{cases}$$

- $\texttt{i} = \texttt{encode}(\texttt{c.nop})$:

$$[\![\texttt{i}]\!](\text{PC},\sigma,\alpha) = \begin{cases} \bot \text{ if } \sigma.\texttt{CCSPROT} \neq 0 \\ \langle \text{PC} + \|\texttt{i}\|, \sigma', \alpha\rangle \text{ otherwise} \\ \quad \sigma'.\texttt{CCS} = \sigma.\texttt{CCS} + \texttt{realign}(\text{PC}, \texttt{u}_{32}(\texttt{i})) \end{cases}$$

The CCS update adds the instruction's opcode to the CCS register, adjusted for PC alignment:[17]

$$\texttt{realign}(\text{PC},\texttt{i}) = \begin{cases} \texttt{i} \text{ if } \text{PC} \equiv 0\ [4] \\ 2^{16}\texttt{LSH}(\texttt{i}) + \texttt{MSH}(\texttt{i}) \text{ otherwise} \end{cases}$$

Trapping instructions include ebreak/c.ebreak which we treat as non-recoverable, and ecall which invokes a syscall whose number is specified in register a7 (x17). We treat an invocation of syscall __NR_exit (1) as program termination.[18]

- $\texttt{i} = \texttt{encode}(\texttt{ebreak})$, $\texttt{i} = \texttt{encode}(\texttt{c.ebreak})$:

$$[\![\texttt{i}]\!](\text{PC},\sigma,\alpha) = \bot$$

- $\texttt{i} = \texttt{encode}(\texttt{ecall})$:

$$[\![\texttt{i}]\!](\text{PC},\sigma,\alpha) = \begin{cases} \texttt{end}(\alpha) \text{ if } \alpha[17] = 1 \text{ and } \sigma.\texttt{CCSPROT} = 0 \\ \bot \text{ otherwise} \end{cases}$$

Among standard instructions, jumps are the most heavily affected. We show here the cases of jal (*jump-and-link*), which in RISC-V is used for unconditional branches and function calls, and beq (*branch-if-equal*), a representative conditional branch. The extension to jalr (which is used to return from functions) is immediate by replacing the target PC + imm of jal with the value of a register in α.

- $\texttt{i} = \texttt{encode}(\texttt{jal}\ \texttt{r}_\texttt{d}, \texttt{imm})$:

[16] As per the RISC-V ISA [*The RISC-V Instruction Set Manual Vol. I* 2019], a 32-bit instruction leader is a value $i : \texttt{u}_{16}$ such that $i \equiv 3\ [4]$ and a 16-bit instruction is any other 16-bit value.

[17] Computing the sum of realigned instruction opcodes ends up being equivalent to summing the values returned by fetch rules in the program, see Lemma 3. We choose to update during decoding because of our intuition that a CCS update during the fetch cycle would be impacted by the fault.

[18] We provide this exit mechanism to have a reasonable program model, but in practice the exit() function is in the non-protected libc, so we don't actually worry about protecting the last block.

$$[\![\mathtt{i}]\!](\mathrm{PC},\sigma,\alpha) = \begin{cases} \bot \text{ if } \sigma.\mathtt{CCSPROT} \neq \mathrm{PC} \\ \langle \mathrm{PC} + \mathtt{imm}, \sigma', \alpha' \rangle \text{ otherwise, with} \\ \quad \sigma'.\mathtt{CCS} = 0 \\ \quad \sigma'.\mathtt{CCSPROT} = 0 \\ \quad \sigma'.\mathtt{CCSD.JO} = 0 \\ \quad \alpha'[\mathtt{r_d}] = \mathrm{PC} + \|i\| + 2 \times \sigma.\mathtt{CCSD.JO} \end{cases}$$

There are two changes compared to traditional jumps: (1) it is only allowed when CCSPROT is set to PC; (2) it accounts for the *jump offset* (JO) field of CCSD to jump over trap barriers after function calls. The jump also prepares the execution of the next block by resetting CCS and other registers to 0. We do not allow the compiler to compress `jal` into the 16-bit version `c.jal` to make sure the next block is also aligned.

Conditional branches are similar, except that if the branch is not taken the block continues and in particular CCS is updated rather than reset.

- $\mathtt{i} = \mathtt{encode}(\mathtt{beq}\ \mathtt{rs}_1, \mathtt{rs}_2, \mathtt{imm})$:

$$[\![\mathtt{i}]\!](\mathrm{PC},\sigma,\alpha) = \begin{cases} \bot \text{ if } \sigma.\mathtt{CCSPROT} \neq \mathrm{PC} \\ \langle \mathrm{PC} + \mathtt{imm}, \sigma', \alpha \rangle \text{ if } \alpha[\mathtt{rs}_1] = \alpha[\mathtt{rs}_2], \text{ with} \\ \quad \sigma'.\mathtt{CCS} = 0 \\ \quad \sigma'.\mathtt{CCSPROT} = 0 \\ \quad \sigma'.\mathtt{CCSD.JO} = 0 \\ \langle \mathrm{PC} + \|\mathtt{i}\|, \sigma', \alpha \rangle \text{ otherwise, with} \\ \quad \sigma'.\mathtt{CCS} = \sigma.\mathtt{CCS} + \mathtt{realign}(\mathrm{PC}, \mathtt{u}_{32}(\mathtt{i})) \end{cases}$$

The last type of instruction is Xccs instructions. Because these have both a 32-bit opcode and a 32-bit argument, they cannot be executed in a single step. A "delay slot" mechanism with σ.CCSDS and the CHECKSUM-DELAY-SLOT rule is used to solve this issue in two steps. At the first step, the Xccs opcode will be fetched and recorded in σ.CCSDS. At the second step, the checksum value will be fetched, and passed as argument to the appropriate semantics function $[\![\sigma.\mathtt{CCSDS}]\!]_{ccs}$.

The first step which records the opcode in CCSDS proceeds as follows:

- $\mathtt{i} = \mathtt{encode}(\mathtt{ccs})$, $\mathtt{i} = \mathtt{encode}(\mathtt{ccsb})$, $\mathtt{i} = \mathtt{encode}(\mathtt{ccscall\ N})$, $\mathtt{i} = \mathtt{encode}(\mathtt{ccscallb\ N})$:

$$[\![\mathtt{i}]\!](\mathrm{PC},\sigma,\alpha) = \begin{cases} \bot \text{ if PC is unaligned or } \sigma.\mathtt{CCSPROT} \neq 0 \\ \langle \mathrm{PC} + \|\mathtt{i}\|, \sigma', \alpha \rangle \text{ otherwise, with} \\ \quad \sigma'.\mathtt{CCS} = \sigma.\mathtt{CCS} + \mathtt{realign}(\mathrm{PC}, \mathtt{u}_{32}(\mathtt{i})) \\ \quad \sigma'.\mathtt{CCSDS} = \mathtt{i} \end{cases}$$

The second step, which always uses the CHECKSUM-DELAY-SLOT rule, reads back from CCSDS and calls the $[\![]\!]_{ccs}$ function by passing the checksum d just fetched from memory as an extra 4th argument. All 4 Xccs instructions have similar semantics; the `-call` variants set `CCSD.JO = N` and the `-b` variants flip the least significant bit of the checksum value before comparing. We factor them as follows.

- Checksum step (d is a checksum value):

$$\begin{aligned} [\![\mathtt{encode}(\mathtt{ccs})]\!]_{ccs} &= \mathtt{check}(0,0) \\ [\![\mathtt{encode}(\mathtt{ccsb})]\!]_{ccs} &= \mathtt{check}(1,0) \\ [\![\mathtt{encode}(\mathtt{ccscall\ N})]\!]_{ccs} &= \mathtt{check}(0,\mathtt{N}) \\ [\![\mathtt{encode}(\mathtt{ccscallb\ N})]\!]_{ccs} &= \mathtt{check}(1,\mathtt{N}) \end{aligned}$$

$$\mathtt{check}(\mathtt{mask} : \mathtt{u}_{32}, \mathtt{JO} : \mathtt{u}_5)(\mathrm{PC},\sigma,\alpha,d) = \begin{cases} \bot \text{ if } \sigma.\mathtt{CCS} \neq (d \oplus \mathtt{mask}) \text{ or } \sigma.\mathtt{CCSPROT} \neq 0 \\ \bot \text{ if } d \text{ is not a valid checksum literal} \\ \langle \mathrm{PC} + 4, \sigma', \alpha \rangle \text{ otherwise, with} \\ \quad \sigma'.\mathtt{CCSPROT} = \mathrm{PC} + 4 \\ \quad \sigma'.\mathtt{CCSDS} = 0 \\ \quad \sigma'.\mathtt{CCSD.JO} = \mathtt{JO} \end{cases}$$

The purpose of the "checksum literals" check is to close attack vectors in which an attacker feeds an incorrect checksum value by injecting a fault between the Xccs opcode and its checksum parameter. We avoid these by preventing a 32-bit value d from appearing verbatim as a checksum value if:

- it decodes as a jump instruction;
- it decodes as an Xccs instruction;
- it decodes as a pair of `c.ebreak`.

The countermeasure gets rid of such values by flipping their LSB, which is shown to be correct in Lemma 9.

B Hardening algorithm and proof of security

B.1 Structure of hardened programs

The security property that we want to establish is that in the execution of a hardened program, any single-fault injection leads to termination with $\perp$ before the end of the current block. This relies on a particular structure for blocks, which we also use to formalize the hardening process.

Definition 5 (Source block)**.**
*A **source block** is a block whose sequence of instructions consists of:*

- *0 or more **straight** (*`add`, `ebreak`, `ecall`*...) or **conditional branch** (*`beq`*) instructions, excluding* `c.ebreak` *and Xccs instructions; followed by*
- *One **unconditional branch instruction** (*`jal`*).*

Unlike classical definitions, **we do not count conditional branches as block terminators**. This improves the correspondence between blocks and checksum regions; in particular, it ensures the invariant that `CCS` resets to 0 at the start of every block.

Definition 6 (Hardened block)**.**
*A **hardened block** is a block* `hb` *obtained by hardening a source block, i.e.* $\mathsf{hb} = \textsc{Harden}(\mathsf{sb}, N)$ *for some source block* `sb`. `HARDEN` *is detailed as Algorithm 2 (which is a copy of Algorithm 1).*

The hardening algorithm processes instructions individually. All the original instructions are kept. Before each jump, a checksum check is added by procedure `addChecksum()`, in the form of an Xccs instruction (`ccscall` for function calls, `ccs` otherwise) followed by the reference checksum value. In order to close certain attack vectors, the checksum value must not decode as a jump or Xccs instruction; if that happens, the LSB of the checksum value is flipped and the Xccs instruction is replaced by its `-b` variant. Finally, a barrier of `c.ebreak` instructions is added at the end.

Now, conditional branches in the middle of a block allow for early exits, but don't reset the checksum. The idea is that upon exiting the block, the expected checksum value is always the sum of all lines from the beginning of the block to the exit instruction. It will nonetheless be useful to split the blocks at each exit point to help formalization.

Lemma 1 (Structure of source blocks)**.**
A source block `sb` *can be uniquely decomposed into* $m \geq 0$ *early sections followed by one final section, i.e.*

$$\mathsf{sb} = \mathsf{se}_1 + \ldots + \mathsf{se}_m + \mathsf{sf} \quad \text{(“+” is concatenation)}$$

where

- *each* se_k *(**source early**,* $1 \leq k \leq m$*) section consists of straight instructions followed by one conditional branch;*

Algorithm 2 Algorithm: `HARDEN`

Input: A source block $[\mathtt{i}_1, \ldots, \mathtt{i}_n]$
Input: Upper bound N for $\mathsf{S}_{32(k)}$ faults each fetch
Output: A hardened block `hb`.

```
hb ← [ ]
sum : u32 ← 0
offset ← 0                                   ▷ Blocks are 4-aligned.
procedure addToBlock(i)
    hb.append(i)
    sum ← sum + realign(offset, u32(i))
    offset ← offset + ‖i‖
procedure addChecksum(i, ib)
    if sum + i is not a valid checksum literal then
        addToBlock(ib)
        addToBlock(sum ⊕ 1)
    else
        addToBlock(i)
        addToBlock(sum)
for i in [i1, ..., in] do
    if i is a jump instruction then
        if offset = 0 then                   ▷ No empty sections.
            addToBlock(encode(nop))
        else if offset ≡ 2 [4] then          ▷ Force alignment.
            addToBlock(encode(c.nop))
        if i is a function call then
            addChecksum(encode(ccscall (2N + 4)),
                        encode(ccscallb (2N + 4)))
        else                                 ▷ Conditional/unconditional jumps.
            addChecksum(encode(ccs), encode(ccsb))
        offset ← 0
    addToBlock(i)
for j = 1 to 2N + 4 do                       ▷ Add trap barrier.
    addToBlock(encode(c.ebreak))
return hb
```

- *the* `sf` *(**source final**) section consists of straight instructions followed by the block's unconditional jump.*

Proof. Because there is exactly one conditional branch per se_k and none other, m must be the number of conditional branches in `sb`. The grouping is straightforward from here. □

Lemma 2 (Structure of hardened blocks)**.**
A hardened block $\mathsf{hb} = \textsc{Harden}(\mathsf{se}_1 + \ldots + \mathsf{se}_m + \mathsf{sf}, N)$ *can be uniquely decomposed into 4-aligned sections*

$$\mathsf{hb} = \mathsf{he}_1 + \ldots + \mathsf{he}_m + \mathsf{hf}$$

where

- *each* he_k *(**hardened early**,* $1 \leq k \leq m$*) section consists of:*
  1. *the straight instructions of* se_k*;*
  2. *an optional* `c.nop` *or* `nop`*, in a way that* 1) *and* 2) *combined are not empty;*

3. *a 4-aligned* `ccs` *or* `ccsb`*;*
4. *a 4-aligned 32-bit checksum value;*
5. *the conditional branch of* $\mathtt{se}_k$*;*

- *the* `hf` ***(hardened final)*** *section consists of:*
  1. *the straight instructions of* `hf`*;*
  2. *an optional* `c.nop` *or* `nop`*, in a way that* 1) *and* 2) *combined are not empty;*
  3. *a 4-aligned* `csscall` *or* `ccscallb` *if the terminator is a function call, a* `ccs` *or* `ccsb` *otherwise;*
  4. *a 4-aligned 32-bit checksum value;*
  5. *the terminator of* `sf`*;*
  6. $2N + 4$ `c.ebreak` *instructions.*

Proof. Decomposition: The main (**for** `i`) loop in `HARDEN` is a morphism for concatenation. Take for $\mathtt{he}_k$/`hf` the hardened sequences corresponding to $\mathtt{se}_k$/`sf` in the input; structurally, we get $\mathtt{hb} = \mathtt{he}_1 + ... + \mathtt{he}_m + \mathtt{hf}$.

Section contents: Straight instructions are clearly preserved by `HARDEN`. For both types of sections, items 2 through 5 are generated by the handling of the section's jump by `HARDEN`. The trap barrier is always added at the end of the block; we can count it towards `hf`.

Alignment: First, all sections have a length that is a multiple of 4. This constraint is ensured by adding a `c.nop` before section's branch if the straight instructions finish on an unaligned address. The following Xccs instruction, checksum value, and branch instruction all occupy 4 bytes each. Then, because the block `hb` itself is 4-aligned and the sections follow each other in memory, each section individually must be 4-aligned. □

Finally, we give an intuitive view of the checksum values by proving that the per-instruction summing process with realignment is equivalent to summing 4-aligned lines. We can do this by comparing both computations on the underlying sequences of $\mathtt{u}_{16}$ values.

Lemma 3 (`realign` sum computes the sum of fetched lines)**.** *Let* $h = (h_0, ..., h_{2n-1}) : \mathtt{u}_{16}^{2n}$ *a sequence of* $\mathtt{u}_{16}$ *("halfwords").*

Assume $(w_0, ..., w_{n-1}) : \mathtt{u}_{32}^{n}$ *is a sequence of* $\mathtt{u}_{32}$ *("words") whose concatenation is h, i.e. such that* $w_i = h_{2i} + 2^{16} h_{2i+1}$*.*

Let $(\mathtt{i}_0, ..., \mathtt{i}_{m-1})$ *a sequence of instructions, and write* $\mathtt{offset}_j = \frac{1}{2}\sum_{k=0}^{j-1} \|i_k\|$ *the offset of instruction j in the sequence (in 16-bit units). Assume the concatenation of this sequence is also h, i.e.* $\mathtt{offset}_m = 2n$ *and for all j,*

- $\mathtt{i}_j = h_{\mathtt{offset}_j}$ *if* $\mathtt{i}_j$ *is a 16-bit instruction;*
- $\mathtt{i}_j = h_{\mathtt{offset}_j} + 2^{16} h_{\mathtt{offset}_j+1}$ *otherwise.*

Then,

$$\sum_{j=0}^{m-1} \mathtt{realign}(2 \cdot \mathtt{offset}_j, \mathtt{u}_{32}(\mathtt{i}_j)) = \sum_{i=0}^{n-1} w_i.$$

Proof. First, rewrite every instruction's realigned contribution in terms of h_i; specifically, we have

$$\forall j,\ \mathtt{realign}(2 \cdot \mathtt{offset}_j, \mathtt{u}_{32}(\mathtt{i}_j)) = \sum_{i=\mathtt{offset}_j}^{\mathtt{offset}_{j+1}-1} 2^{16(i \bmod 2)} h_i.$$

This we can show by distinguishing four cases:

- $\mathtt{i}_j$ is 16-bit, $\mathtt{offset}_j \bmod 2 = 0$:
  - LHS is $\mathtt{i}_j$
  - RHS is $h_{\mathtt{offset}_j} = \mathtt{i}_j$
- $\mathtt{i}_j$ is 16-bit, $\mathtt{offset}_j \bmod 2 = 1$:
  - LHS is $2^{16}\mathtt{i}_j$
  - RHS is $2^{16} h_{\mathtt{offset}_j} = 2^{16}\mathtt{i}_j$
- $\mathtt{i}_j$ is 32-bit, $\mathtt{offset}_j \bmod 2 = 0$:
  - LHS is $\mathtt{i}_j$
  - RHS is $h_{\mathtt{offset}_j} + 2^{16} h_{\mathtt{offset}_j+1} = \mathtt{i}_j$
- $\mathtt{i}_j$ is 32-bit, $\mathtt{offset}_j \bmod 2 = 1$:
  - LHS is $\mathtt{MSH}(\mathtt{i}_j) + 2^{16}\mathtt{LSH}(\mathtt{i}_j)$
  - RHS is $2^{16} h_{\mathtt{offset}_j} + h_{\mathtt{offset}_j+1} = 2^{16}\mathtt{LSH}(\mathtt{i}_j) + \mathtt{MSH}(\mathtt{i}_j)$

Now, by summing from $j = 0$ to $m - 1$, we get

$$S := \sum_{j=0}^{m-1} \mathtt{realign}(\mathtt{offset}_j, \mathtt{u}_{32}(\mathtt{i}_j)) = \sum_{i=0}^{\mathtt{offset}_m-1} 2^{16(i \bmod 2)} h_i.$$

Recalling that $\mathtt{offset}_m = 2n$, this equivalent to unfolding w_i into a sum of $\mathtt{u}_{16}$:

$$\sum_{i=0}^{n-1} w_i = \sum_{i=0}^{2n-1} 2^{16(i \bmod 2)} h_i = S.$$

□

B.2 Program state upon leaving a hardened block

We will now show that the counter-measure ensures good properties in the execution of hardened blocks, namely that they can only be exited by a legitimate jump instruction and while validating the checksum associated to that jump.

In the following, we assume a program P and a successful execution $e = [s_1, ..., s_{|e|}]$, with each step being written

$$s_i = \langle \mathrm{PC}_i, \rho_i, \sigma_i, \alpha_i \rangle \rightarrow r_i,$$

which structurally implies that

$$\begin{cases} r_i = \langle \mathrm{PC}_{i+1}, \rho_{i+1}, \sigma_{i+1}, \alpha_{i+1} \rangle & \text{if } i < |e|; \\ r_{|e|} = \mathsf{end}(\alpha_{|e|}). \end{cases}$$

The first step in the proof is to show that the trap barrier prevents the end of a hardened block from being reached, meaning that any exit must happen through a jump instruction.

Lemma 4 (Hardened blocks must be exited by jumps)**.**

Let `hb` *be a hardened block of P, and assume the execution enters* `hb`*, meaning that there exists t ("top") such that* $PC_t = \mathtt{blockAddr(hb)}$*.*

Let n be the number of instructions executed without leaving `hb`*, i.e. the largest number such that the following all hold:*

1. $t + n \leq |e|$*;*
2. $\forall i \in [t, t+n)$, $PC_{i+1} \in$ `blockSpan(hb)`*;*
3. $\forall i \in [t, t+n)$, s_i *is not a jump instruction.*[19]

Then, either

- $t + n = |e|$ *(hence* s_{t+n} *is a successful termination by an invocation of the* `exit` *syscall), or*
- $t + n < |e|$ *and* s_{t+n} *is a jump instruction.*

Proof. By contradiction. An exit not covered by the lemma statement must be with $t + n < |e|$ and s_{t+n} not a jump instruction. As such, there must be no jump instruction in the entire sub-sequence $s_t \ldots s_{t+n}$.

For a non-jumping step s_i, we always have $\mathrm{PC}_{i+1} > \mathrm{PC}_i$. This is because the only two changes to PC are the update by the instruction's semantics (which always adds $\|\mathtt{i}\| > 0$) and the potential skip from the S32(k) fetch rule (which is $4k \geq 0$).

Therefore, $(\mathrm{PC}_i)_{t \leq i \leq t+n}$ is a strictly increasing sequence, which means that every address in `blockSpan(hb)` is part of exactly one $[\mathrm{PC}_i, \mathrm{PC}_{i+1})$ interval (i.e. there is exactly one step that consumes it as part of its execution).

So then, we can look at `hb`'s ending, which has the following 4-aligned sequence (where each line covers 4 bytes of code):

```
    ccs/ccsb/ccscall/ccscallb
    <checksum value>
    j32
L:  c.ebreak; c.ebreak # (repeat N+2 times)
```

The `j32` marker represents the jump instruction, which is always a 32-bit instruction. The trap barrier starts at address `L` and contains $2N + 4$ `c.ebreak` instructions, therefore the block ends at $\mathtt{L} + 4N + 8$.

From the previous argument, there is exactly one step s_i such that address `L` lies in the interval $[\mathrm{PC}_i, \mathrm{PC}_{i+1})$, meaning that the line at address `L` is loaded or skipped during the execution of step s_i. Step s_i starts at PC_i, which can be written in a unique way as either $\mathrm{PC}_i = \mathtt{L} - 4k$ or $\mathrm{PC}_i = \mathtt{L} - 4k - 2$ depending on its alignment.

Now by analyzing all cases for the value of k and alignment of PC, we can show that the execution always crashes at or shortly after s_i, completing the contradiction argument.

- $k > 0$. This case is only possible if step s_i uses the fetch rule S32(k') because $\mathrm{PC}_{i+1} > \mathtt{L}$, yet all other options for non-jump instructions result in $\mathrm{PC}_{i+1} \leq \mathrm{PC}_i + 4$. Since $k' \leq N$, the fetch will not reach beyond the trap barrier, so the fetch returns a pair of `c.ebreak` instructions. These then get consumed by s_i's step rule, which we can identify from the alignment of PC.
  - PC aligned: s_i must then be using either ALIGNED-16 or CHECKSUM-DELAY-SLOT. In the first case, the first `c.ebreak` fetched by s_i is executed, leading to a trap. In the second case, the checksum check also traps because double `c.ebreak` is not a valid checksum literal.
  - PC unaligned: since there was a fetch, UNALIGNED-32 is being used. A 32-bit instruction is composed from $\mathtt{MSH}(\rho_i)$ and the first `c.ebreak`. Recall that s_i is not a jump, so this instruction either crashes or leads into s_{i+1} with PC_{i+1} unaligned. In this case, s_{i+1} will use UNALIGNED-16 and run the second `c.ebreak`, trapping as claimed.
- $k = 0$. In this case, PC_i must be either `L` or $\mathtt{L} - 2$.
  - $\mathrm{PC}_i = \mathtt{L}$ (aligned). Here, s_i can either use rule ALIGNED-16, ALIGNED-32, or CHECKSUM-DELAY-SLOT, all of which require a fetch that can only return $\rho_i =$ `encode(j32)` (if S&R32 is used) or a pair of `c.ebreak`. Using ALIGNED-16 and ALIGNED-32 would either crash from running a `c.ebreak` or contradict the hypothesis that s_i is not a jump. Using CHECKSUM-DELAY-SLOT would also crash because neither fetch result is a valid checksum literal.
  - $\mathrm{PC}_i = \mathtt{L} - 2$ (unaligned). Because there is a fetch, s_i must be using rule UNALIGNED-32. Much like the similar case in $k > 0$, the instruction recomposed from $\mathtt{MSH}(\rho_i)$ and $\mathtt{LSH}(d)$ cannot be a jump, so if it doesn't crash execution continues to s_{i+1} with the new $\mathtt{LSH}(\rho_{i+1})$.

    If s_i uses fetch rule NOFAULT or S32(k'), then ρ_{i+1} is a pair of `c.ebreak`, therefore s_{i+1} will use UNALIGNED-16 and crash running the second `c.ebreak`.

    Execution can only proceed past s_{i+1} if s_i uses fetch rule S&R32, in which case s_{i+1} is a repeat of s_i 4 bytes later ($\mathrm{PC}_{i+1} = \mathtt{L} + 2$). The analysis of s_i can be repeated, with two changes. First, S32(k') could now reach 4 bytes further, up to $\mathtt{L}+4+4N$, which hits the last 4 bytes of the barrier. This shows why $2N + 4$ `c.ebreak` are required. Second, using S&R32 no longer succeeds because ρ_{i+1} is now also a pair of `c.ebreak`. Therefore, s_{i+2} crashes.

This guaranteed crash implies that a successful execution can only exit a protected block in one of the ways described by the theorem statement. □

The next objective is to show that the XCCS instruction, checksum and jump instruction sequence is secure in that only *legitimate* jumps can be used to exit a hardened block. A key part of this is that only original instructions must be executed around the time of the jump, not forged instructions. This synchronization is guaranteed by the fact that an XCCS instruction is needed to jump, and XCCS instructions *cannot* be forged.

Lemma 5 (XCCS instructions can be repeated but not forged)**.** *Let* `hb` $= [\mathtt{i}_1, \ldots, \mathtt{i}_{|\mathtt{hb}|}]$ *be a hardened block and*

$$s = \langle PC, \rho, \sigma, \alpha \rangle \rightarrow \langle PC', \rho', \sigma', \alpha' \rangle$$

a step that successfully runs an XCCS opcode within `hb`*, i.e. such that* $[PC, PC') \subseteq$ `blockSpan(hb)`*. Further assume that we don't repeat upon landing in* `hb`*, i.e. either* s *doesn't use rule* S&R32 *or* $PC \geq$ `blockAddr(hb)` $+ 4$*.*

[19] For now we consider a forged jump that jumps back somewhere into the block as "leaving" the block; we will later show that this cannot happen.

Then, the instruction executed by s is an original instruction of hb*, in the sense that there is an Xccs instruction* $\mathtt{i}_j \in \mathtt{hb}$ *(with address* $\mathtt{L} = \mathtt{blockAddr(hb)} + \sum_{k=0}^{j-1} \|\mathtt{i}_k\|$*) such that one of the following is true:*

1. *s uses fetch rule* NoFault *and* $PC = \mathtt{L}$*;*
2. *s uses fetch rule* S32(k) *and* $PC = \mathtt{L} - 4k$*;*
3. *s uses fetch rule* S&R32 *and* $PC = \mathtt{L} + 4$*.*

Proof. Xccs instructions trap when executed with unaligned PC, so they can only be run by step rule Aligned-32. This rule always uses the value d returned by a fetch as an opcode, which is a 4-aligned value found in program memory at an address that depends on the fetch rule:

- With NoFault, d is the value at PC;
- With S32(k), d is the value at PC + $4k$;
- With S&R32, d is the value at PC − 4.

Given the hypotheses and the length of the trap barrier, this value must be a 4-aligned value within hb, which can intersect the instruction sequence in three ways:

1. d matches a 32-bit instruction $\mathtt{i}_j \in \mathtt{hb}$: then the lemma is obviously true.
2. LSH(d) is the last two bytes of an instruction $\mathtt{i}_j \in \mathtt{hb}$ and MSH(d) is the first two bytes of $\mathtt{i}_{j+1}$. This is impossible because d is an Xccs opcode so MSH(d) = 0. The 16-bit zero value is reserved in the RISC-V ISA as an invalid opcode, thus $\mathtt{i}_{j+1}$ could not be a valid instruction.
3. d matches a 32-bit checksum argument to an Xccs instruction: this is also impossible because Xccs opcodes are not valid checksum literals.

Since only case 1) is possible, s_i must indeed be executing a legitimate instruction $\mathtt{i}_j \in \mathtt{hb}$ (maybe after a fault). □

Definition 7 (Legitimate entry)**.**
A state $\langle PC, \rho, \sigma, \alpha \rangle$ is called a ***legitimate entry*** *into a block* bb *if* $PC = \mathtt{blockAddr(bb)}$, $\sigma.\mathtt{CCSPROT} = \sigma.\mathtt{CCSDS} = 0$*, and ρ does not decode as an Xccs instruction.*

Definition 8 (Legitimate execution of a block)**.**
A sequence of steps $s_t \dots s_b$ ("top", "bottom") is defined as a ***legitimate execution of a hardened block*** **hb** *if the following conditions are met:*

- *s_t's initial state is a legitimate entry into* hb*;*
- $\forall i \in [t, b)$, $PC_{i+1} \in \mathtt{blockSpan(hb)}$*;*
- *s_b is the first and only instruction in the sequence to be either a taken jump or an invocation of the* `exit` *syscall.*

Lemma 6 (Jumps out of hardened blocks must be legitimate)**.**
Let $\mathtt{hb} = [\mathtt{i}_1, \dots, \mathtt{i}_{|\mathtt{hb}|}]$ *a hardened block of P, and $s_t \dots s_b$ a legitimate execution of* hb *where s_b is a jump. Then:*

1. *$b \geq t + 2$ and s_{b-2} is an Xccs instruction;*
2. *The jump is legitimate, i.e. there is an instruction* $\mathtt{i}_j \in \mathtt{hb}$ *such that* $PC_b = \mathtt{blockAddr(hb)} + \sum_{k=0}^{j-1} \|\mathtt{i}_k\|$ *and step s_b executes the opcode* $\mathtt{i}_j$*.*
3. *The checksum is correct when leaving the block, i.e.*

$$\sigma_b.\mathtt{CCS} = \begin{cases} [PC_b - 4] & \text{if it's a valid checksum literal;} \\ [PC_b - 4] \oplus 1 & \text{otherwise.} \end{cases}$$

4. *s_b's final state is a legitimate entry into another block.*

Proof. Walking back from the last few instructions, for s_b to be a jump and not trap, we must have $\sigma_b.\mathtt{CCSPROT} \neq 0$. Therefore $b > t$ and s_{b-1} must use the Checksum-Delay-Slot rule, as no other type of step can *end* with CCSPROT ≠ 0. This implies that $\sigma_{b-1}.\mathtt{CCSDS} \neq 0$, so once again $b - 1 > t$ and s_{b-2} must execute an Xccs instruction, because CCSDS can only be non-zero for one step and no other instruction sets it.

By Lemma 5, there must be an original Xccs instruction from hb being executed by s_{b-2}, and such instructions are only found at the end of early or final sections, which have the following structure:

```
L:     ccs/ccsb/ccscall/ccscallb
L+4:   <checksum value>
L+8:   j32
L+12:  # ... next section or c.ebreak ...
```

Note that while s_{b-2} runs the opcode found in memory at address L, PC_{b-2} is not guaranteed to be L since faults might be involved. The lemma focuses on showing that even then, in all execution scenarios starting from s_{b-2} the legitimate jump at L + 8 must be taken by s_b while also passing the checksum at L + 4.[20]

First note that, as Xccs instructions set CCSPROT = PC + 8 and both s_{b-2} and s_b already increment PC by 4, neither s_{b-1} nor s_b can fetch with S32(k) as that would cause PC > CCSPROT in s_b and trap. The only variable after s_{b-2} is whether the S&R32 fetch rule is used to change either fetched value.

- If s_{b-2} uses either of the fetch rules NoFault and S32(k), then $PC_{b-1} = \mathtt{L} + 4$ and $PC_b = \mathtt{L} + 8$.

  Then, s_{b-1} uses the Checksum-Delay-Slot rule after fetching a checksum literal. Attacking the fetch with S&R32 would return the Xccs opcode of s_{b-2}, and trap because it's not a valid checksum literal. Thus s_{b-1} must fetch with NoFault, passing the checksum.

  Finally, s_b performs the jump. Fetching with S&R32 is again impossible as that would execute the checksum literal, which cannot be a jump. Therefore, s_b fetches with NoFault, and executes the intended jump at L + 8 while passing the intended checksum.
- If s_b uses the fetch rule S&R32, then $PC_b = \mathtt{L} + 4$, which means that $PC_{b-1} = \mathtt{L} + 8$ and $PC_b = \mathtt{L} + 12$.

  This time, s_{b-1} must fetch with S&R32 to obtain the intended checksum, as a NoFault fetch would return the jump opcode, which is not a valid checksum literal.

[20] The analysis in this lemma remains correct even if rule S32(k) caused the execution of extra nop instructions, as that would reset CCSPROT to 0 and just prevent the jump altogether.

Then, s_b must also fetch with S&R32 to get the jump, since a NoFault fetch would the value at L + 12, which cannot be a jump.

Notice that in the second case the entire jump widget is "delayed" by 4 bytes by repeated attack with S&R32. This is still a legitimate exit for the purpose of this lemma. We will show later that in single-fault cases, the checksum would not pass in such a scenario.

Now, since the jump was executed properly, it is fairly easy to show that s_b is a legitimate entry into another block.

- The target of the jump is the `blockAddr` of another block by assumption about the validity of the entire program P.
- After the jump, we have $\sigma_b.\texttt{CCSPROT} = 0$ and $\sigma_b.\texttt{CCSDS} = 0$ as a result of the semantics of s_{b-1} and s_b.
- Finally, ρ_{b+1} is either a jump, a `c.ebreak`, or the first line of the next section (which is never an Xccs instruction).

□

B.3 Security guarantees

This key lemma implies that legitimate entries and exits from hardened blocks is an execution invariant. We can sum this up in the most general (multi-fault) case as every block guaranteeing its checksum must pass.

Theorem 1 (Security guarantee for multi-fault executions)**.**

Let P a fully hardened program and $e = [s_0, \ldots, s_{|e|}]$ an execution such that

- *s_0 is a legitimate entry into a block of P;*
- *$s_{|e|}$ ends successfully, returning some $\mathsf{end}(\alpha)$.*

Then there exists a sequence $[\mathsf{hb}_1, \ldots, \mathsf{hb}_m]$ of blocks of P such that

1. *The execution e can be partitioned into subsequences $(s_{t_i} \ldots s_{b_i})_{1 \le i \le m}$ each a legitimate execution of hb_i;*
2. *Each s_{b_i} $(i \neq m)$ is a legitimate jump of hb_i and $\sigma_{b_i}.\texttt{CCS}$ is the correct checksum associated with that jump.*

Proof. By induction, constructing (t_i), (b_i) and (hb_i) along the way.

Assume s_{t_i} is a legitimate entry into hb_i (which is true for $i = 0$). By Lemma 4, there is b_i such that $s_{t_i} \ldots s_{b_i}$ is an execution of hb_1, with s_{b_i} leaving either by successful termination or by a jump. If s_{b_i} ends the program, we're done. Otherwise, by Lemma 6, s_{b_i} is a legitimate jump out of hb_i with $\sigma_{b_i}.\texttt{CCS}$ passing the associated checksum, and its final state is a legitimate entry into another block hb_{i+1}. Define $t_{i+1} := b_i + 1$ and start over. □

As we've discussed previously, this doesn't completely rule out attacks, because for some blocks there exist multi-faulted paths whose checksum collides with the expected checksum. However, such paths do not exist when a single fault is injected during the execution of a block. To show this, we can go back to the sum-of-u_{32} expression of the checksum and reason on the series of fetches that builds it.

Definition 9 (Fetch trace)**.**
*The **fetch trace** of a subsequence $s_t \ldots s_b$ of e is the tuple*

$$(N, S, R, \rho_{\mathsf{entry}}, (a_i), (k_i), \mathsf{next})$$

where fetches are partitioned by rule type:

- $N = \{i \mid s_i$ *uses the* NoFault *rule*$\}$
- $S = \{i \mid s_i$ *uses the* S32(k) *rule*$\}$
- $R = \{i \mid s_i$ *uses the* S&R32 *rule and* $i \neq t\}$
- $\rho_{\mathsf{entry}} = \rho_t$ *if s_t uses the* S&R32 *rule, 0 otherwise*

and relevant information is recorded as follows:

- *for $i \in N \cup S \cup R$, a_i is the address fetched by s_i;*
- *for $i \in S$, k_i is the parameter to* S32(·) *(0 for other i);*
- `next` *gives the next step that includes a fetch, i.e.*

$$\mathsf{next}(i) = \mathtt{min}\ \{j \in N \cup S \cup R \mid j > i\}.$$

Lemma 7 (Relation between checksum and fetch trace)**.**
Let hb *a hardened block of P, $s_t \ldots s_b$ a legitimate execution of* hb *and $(N, S, R, \rho_{\mathsf{entry}}, (a_i), (k_i), \mathsf{next})$ the fetch trace of $s_t \ldots s_{b-2}$ (only up to the last Xccs instruction that counts towards the checksum).*

Then fetched addresses are contiguous in the sense that

$$\forall i < b - 2, a_{\mathsf{next}(i)} = a_i + 4k_i + 4$$

and the checksum upon leaving the block is

$$\sigma_{b-1}.\texttt{CCS} = \sum_{i \in N} [a_i] + \sum_{i \in S} [a_i + 4k_i] + \sum_{i \in R} [a_i - 4] + \rho_{\mathsf{entry}}.$$

Proof. The checksum upon leaving the block is $\sigma_b = \sigma_{b-1}$ (since CCS is not updated during checks and jumps). By definition of instructions' semantics, σ_{b-1} is the realigned sum of the instructions executed by $s_t \ldots s_{b-2}$ (i.e. the values being passed to $[\![\cdot]\!]$ or $[\![\cdot]\!]_{ccs}$ in each step rule).

Because the sequence takes no branch, each instruction `i` ends with PC ← PC + ‖i‖ so each new fetch queries the data immediately following the previous fetch. Accounting for skips in S32(k) means that $a_{\mathsf{next}(i)} = a_i + 4k_i + 4$ (k_i being 0 for other rules).

In addition, the sequence starts on a 4-aligned boundary and ends with a 4-aligned Xccs instruction, so the concatenation of executed instructions' encodings is exactly equal to the concatenation of values returned by fetches. Thus, by Lemma 3, $\sigma_{b-1}.\texttt{CCS}$ is equal to the sum of values returned by fetches. These can be determined for all four categories of fetches:

- For $i \in N$, NoFault returns $[a_i]$;
- For $i \in S$, S32(k) returns $[a_i + 4k_i]$;
- For $i \in R$ $(i > t)$, S&R32 returns ρ_i, which is always equal to $[a_i - 4]$ (because $\rho_i = [a_i]$ at the end of every step that includes a fetch);

- If s_t uses the S&R32 rule, then the fetch for s_t returns ρ_t (the last value fetched by the previous block), otherwise it's counted by N and S.

Adding these up yields the claimed formula for σ_{b-1}.CCS. □

Lemma 8 (Single faults always invalidate the checksum)**.** *Let* hb *a hardened block of P and $s_t \dots s_b$ a legitimate execution of* hb *ending in a jump. Assume the maximum number of skips allowed in a single* S32(k) *rule is $N = 1$. Then there cannot be exactly one fault attack during the execution of the block, i.e.*

$$\mathsf{Card}\ \{i \in [t, b] \mid s_i \text{ uses } \mathsf{S32}(k) \text{ or } \mathsf{S\&R32}\} \neq 1.$$

Proof. By Lemma 6, a legitimate exit by a jump requires an Xccs instruction at s_{b-2}, and s_{b-1} and s_b must either both use NoFault or both use S&R32. The second option immediately implies the theorem; this leaves the first.

Let $(N, S, R, \rho_{\mathsf{entry}}, (a_i), (k_i), \mathtt{next})$ be the fetch trace of $s_t \dots s_{b-2}$. Because s_{b-1} and s_b both use NoFault, the fault attack must occur in $s_t \dots s_{b-2}$, which leaves 3 options.

- s_t uses S&R32: by Lemma 7, the checksum at the end of the block is
$$\sigma_{b-1}.\mathsf{CCS} = \sum_{i=t+1}^{b-2} [a_i] + \rho_{\mathsf{entry}},$$
with $a_i = \mathtt{blockAddr(hb)} + 4i$. Remember that the expected checksum is
$$\sigma_{\mathsf{expected}} = \sum_{i=t}^{b-2} [\mathtt{blockAddr(hb)} + 4i].$$
The difference is $\rho_{\mathsf{entry}} - [\mathtt{blockAddr(hb)}] = \rho_t - [a_t]$, which is non-zero due to the condition on S&R32 (silent replacements do not count as faults). Therefore the checksum doesn't pass, contradicting the hypothesis that $s_t \dots s_b$ is a legitimate execution of hb.
- Some s_j uses S&R32 ($j \neq t$): by Lemma 7, the checksum is
$$\sigma_{b-1}.\mathsf{CCS} = \sum_{i=t}^{b-2} [a_i] - [a_j] + [a_j - 4]$$
still with $a_i = \mathtt{blockAddr(hb)} + 4i$. The difference with $\sigma_{\mathsf{expected}}$ is $[a_j - 4] - [a_j] = \rho_j - [a_j]$ which is again non-zero due to the condition on S&R32.
- Some s_j uses S32(1): still by Lemma 7, the checksum is now
$$\sigma_{b-1}.\mathsf{CCS} = \sum_{i \in N} [a_i] + [a_j + 4] = \sum_{i=t+1}^{b-2} [a_i] - [a_j],$$
since the execution is offset by the skip at s_j, leading to $a_i = \mathtt{blockAddr(hb)} + 4i + 4(i \geq j)$. The difference is $[a_j]$; for the checksum to pass we must have $[a_j] = 0$. This is impossible: for the same reasons as discussed in Lemma 5, a 4-aligned zero value cannot intersect any instructions, so it would have to be a checksum value... but then s_j would crash. This is because the execution was not faulted up to this point, so s_j must be using rule Checksum-Delay-Slot with S32(1). As a result, s_j fetching $[a_j + 4]$, which is not a valid checksum literal (it's a 32-bit jump), leads to a crash. □

We are now finally able to prove the security property that a successful execution with no more than one fault per block has no faults at all.[21]

Theorem 2 (Security guarantee for single-fault executions)**.**

Let P a fully hardened program and $e = [s_0, \dots, s_{|e|}]$ an execution such that

- *s_0 is a legitimate entry into a block of P;*
- *$s_{|e|}$ ends successfully, returning some* $\mathsf{end}(\alpha)$.

Let $[\mathsf{hb}_1, \dots, \mathsf{hb}_m], (s_{t_i} \dots s_{b_i})_{1 \leq i \leq m}$ be the partition of e into block executions outlined by Theorem 1.

If each segment $s_{t_i} \dots s_{b_i}$ uses at most one faulted fetch rule and the last segment $s_{t_m} \dots s_{b_m}$ is faultless, then e is faultless.

Proof. By Theorem 1, segments $i = 1$ to $m - 1$ all validate their checksums, and by hypothesis, they use at most one faulted fetch rule. By Lemma 8, they must in fact use no faulted fetch. The last block is itself faultless by hypothesis. As a result, the entire execution is legitimate. □

B.4 Feasibility of Algorithm Harden

Algorithm 2 relies on the ability to get rid of invalid checksum literals by flipping their Least Significant Bit (LSB). We still have to show that this indeed works.

Lemma 9 (Invalid checksum literals can be avoided)**.** *If $d : \mathsf{u}_{32}$ is an invalid checksum literal, then $(d \oplus 1) + 2^{14}$ is a valid checksum literal.*

Proof. Invalid checksum literals are jumps, Xccs instructions, and pairs of `c.ebreak`. The low 7 bits of each invalid literal is listed in Table **??**; note how flipping the LSB never yields a new pattern that's present in the table.

It is important that the validity of the checksum literal can be inferred from the low 7 bits only. This is because flipping the LSB of the checksum literal also comes with an update to the Xccs opcode (replacing `ccs` by `ccsb` or `ccscall` by `ccscallb`). This update amounts to flipping bit 14 of the opcode, thus affecting bits 14–31 of the checksum value. Since bits 0–6 are not affected, the final checksum literal is still valid even after accounting for this change. □

[21] We exclude the last block since we don't protect `ecall`; in practice it's in non-protected libc and the entry into the `exit()` function is a function call that is itself protected.