

SHIELD against RowHammer Bitflips

Prachi Ingle (pi928)

Sid Sabhnani (sks3897)

CS 380S Final Project

1 PROJECT GOALS

Attacks such as Rowhammer [9] demonstrate that even strong static guarantees can be undermined at runtime. For example, software-based fault isolation (SFI) [10], [11] methods to sandbox untrusted code rely critically on the integrity of their modifications to the object code of a dis-trusted module; they are easily bypassed in the presence of Rowhammer exploits. To restrict the critical gap between static assurances and real-world execution behavior, we seek to develop a scheme that assures security against an adversary that can perform a bitflip in any instruction.

Practically, our project centers around modifying the compilation process of a given binary to emit additional instructions that mitigate against bitflips without changing the intended program flow and keeping overhead minimal. We target the RISC-V architecture and assume a maximum of 1 bitflip can occur per instruction.

2 BACKGROUND

2.1 Adversary Model

We assume the following adversary model, depicted in Figure 1. In this model, the user will submit their code to an untrusted executor (i.e, an untrusted cloud provider) that can perform up to 1 bitflip per instruction. Ideally, if the provider performed a bitflip, then the program should crash when the provider attempts to run the bitflipped program. We assume the adversary has a limited control over the execution environment—such as a compromised cloud provider—and can induce up to one bitflip per 32-bit instruction in the binary. This models real-world Rowhammer-style attacks, where an attacker exploits physical memory vulnerabilities to flip individual bits without software-level access. The submitted code is thus executed in an untrusted environment capable of flipping a single bit in any instruction. Our goal is to ensure that any such modification results in a safe failure—ideally, a program crash—rather than allowing unintended behavior or privilege escalation.

This model is realistic in scenarios where businesses deploy sensitive computations to cloud platforms, which may handle confidential data like API keys or customer data. If an attacker gains low-level access to the cloud provider’s hardware, they could launch targeted bitflip attacks. The code must protect against a universal Rowhammer gadget at the granularity of 1 bitflip per 32 bits.

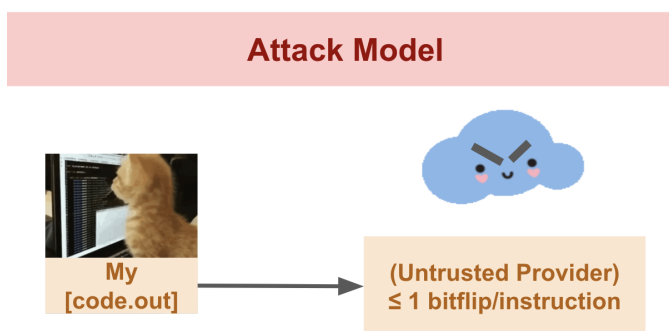


Fig. 1: Our assumed attack model

2.2 Software we Modify

To this end, we worked on two mitigation approaches this semester: (1) Modifying TinyC compiler to emit additional instructions that fault when a bitflip is detected and (2) Adding a few steps to the compilation and linking process to ensure user binaries cryptographically (thus, statistically) match at compile time what they were at runtime. We dub this latter approach **SHIELD (Secure Hashing Integrity Execution Linked Defense)** and find this approach works best. Our code is found in the two repos described below.

2.2.1 Approach 1: Modify Tiny C Compiler

The Tiny C Compiler is an open-source compiler for C and assembly programs that can target x86, ARM, and RISC-V Architecture. For this project, we focus on modifying how TinyC generates binaries for the RISC-V architecture. Specifically, we modify `tinycc/riscv64-asm.c` to emit additional instructions that protect against Rowhammer [2]. Our changes are available on GitHub¹.

2.2.2 Approach 2: SHIELD

The code for this approach is entirely self-contained and available on GitHub². We implement SHIELD by modifying the linking process to include a constructor that verifies the integrity of the binary at runtime. To use SHIELD, users simply compile and link their program against our verification code. The full pipeline is as follows:

- 1) **Start with any user program** (e.g., `main.c`).
- 2) **Compile and link using our custom setup** — Our Makefile builds the user program alongside our `verify.c` file, using a custom linker script `layout.ld`. The `verify.c` file defines a **constructor function**, which executes **before main** when the binary is loaded. This constructor computes a runtime hash of the program’s `.text` section and compares it against a compile-time hash embedded via macros during linking. If the hashes do not match, indicating a possible bitflip or tampering, the program safely aborts before any user code is run.

Ultimately, in this approach, we contribute a new way of building and linking a program the user wishes to protect in the given attack model. We will provide further details and justification of this approach in Section 3.

2.3 Previous Security Concerns: Rowhammer

As DRAM cells become smaller and closer together, repeatedly accessing a memory location on a DRAM module’s row can cause the charge to leak from the cell capacitor and affect another row. With enough targeted accesses, this can cause a target cell to flip its bit. Attackers can thus leverage these unintended physical effects to flip important bits in DRAM, violating the hardware’s policy that changes to a memory location should not affect another. Google’s Project Zero demonstrated the following Rowhammer-based attacks: (1) breaking out of a NaCl sandbox and calling host OS syscalls directly and (2) gaining kernel access to all physical memory by causing a bitflip in a page table entry [9].

1. <https://github.com/sidsabh/tinycc>
2. <https://github.com/sidsabh/rowhammer-patch>

2.4 Related Work: Previous Rowhammer Mitigations

Many Rowhammer hardware-based mitigation strategies center around increasing the refresh rate (how often the DRAM capacitor cells are refreshed with their correct value), using hardware counters to identify which rows should be refreshed, or using error-correcting codes to undo random bitflips [6]. ZebRam introduces a software-based approach that surrounds every used DRAM row with guard rows to absorb Rowhammer-induced bitflips [6]. However, in our adversary model, we abstract the Rowhammer exploit to be any exploit where an attacker can perform one bitflip per instruction. This is relevant as we lower precision with faster CPUs or if another bug like Rowhammer appears without reasonable prevention [3].

Such an adversary has already been defined and well-studied in the case of 1 flip in an entire program: single-event upset (SEU). The field of “software fault tolerance” addresses this adversary with seminal papers like SWIFT [8]. They manually analyze an ISA to instrument binaries with possible bitflips using techniques like duplication and hash-based instrumentation.

Other work [5] suggests an interesting area of future work would be to develop a compiler that guarantees at least N bitflips are required to alter program flow [1][4]. Our paper explores this idea in approach one using the compiler and in approach two using linking.

A quick point on error-correcting codes: Hamming codes and related ideas mitigate transmission issues on data (e.g., 6 bits for 32 bits Hamming Code), but require a trusted compute to determine the issue. Therefore, they are unrelated to this project as we seek to ensure integrity of computation, not data.

3 OUR APPROACH

3.1 Tools

3.1.1 Infrastructure

We conducted our experiments on one of the authors’ personal desktop machines, whose specifications are listed in Table 1. We chose not to use the UTCS lab machines for the following reasons:

- 1) **Lack of administrative privileges:** Without `sudo` access, we were unable to install necessary dependencies or tools via `apt`.
- 2) **Memory limitations for Rowhammer attacks:** The lab machines use DDR4 memory, which is significantly more resistant to Rowhammer-style bitflips. For instance, when running the [Google Rowhammer test](#), our desktop with DDR3 memory succeeded in under 90 seconds, whereas the lab machines never succeeded. We never ended up using Rowhammer as we switched to manual bitflips due to an abstracted adversary.
- 3) **Need for an isolated and configurable environment:** Our setup allowed for full system control, including root access and kernel tuning, which was necessary to test our compiler protections in a realistic threat model.

To enable remote testing, we configured the desktop as an SSH-accessible server by registering it with a Dynamic DNS provider and creating a dedicated user account with the appropriate permissions, resource quotas, and storage limits.

Field	Description
Architecture:	x86_64
CPU(s):	8
Thread(s) per core:	2
RAM type:	24 GB DDRM 3
CPU Model Name	Intel(R) Core(TM) i7-3770K CPU @ 3.50GHz
OS Version	Ubuntu 24.04.1 LTS
qemu-riscv64 version	8.2.2 (Debian 1:8.2.2+ds-0ubuntu1.6)

TABLE 1: Hardware Details. CPU Information: `lscpu`. OS Information: `lsb_release -d`. Memory details `sudo dmidecode --type memory`

3.1.2 Approach 1: QEMU Emulator and RISC-V Cross Compiler

Though we are targeting the RISC-V architecture, our desktop is an x86_64 machine (Table 1). Neither of us have access to a RISC-V machine. Thus, we configured the Tiny C Compiler to cross-compile RISC-V assembly. We then run RISC-V assembly on our x86_64 machine by using QEMU – an emulator that can emulate the RISC-V architecture [7]. We have Debian RISC-V `qcow2` image that we can boot into for OS-level support (`qemu-system-riscv64`, but we only ended up working on application-level (`qemu-riscv64`).

3.1.3 Approach 2: Sysroot, Build System

To enable cross-compilation of RISC-V binaries that depend on external libraries such as OpenSSL, LibELF, and zlib, we configured a RISC-V sysroot. This involved installing RISC-V versions of the required libraries and setting several environment variables to ensure both the compiler and runtime linker could locate the appropriate files.

Specifically, we set environment variables `C_INCLUDE_PATH`, `LIBRARY_PATH`, and `LD_LIBRARY_PATH` to point to the RISC-V include and library directories. These paths ensured that header files were found during compilation and that shared libraries were correctly resolved during emulation.

We developed a custom build pipeline that automates the process of compiling, hashing, and validating RISC-V binaries against bitflips. Given a user program (e.g., `tests/simple.c`), our build system performs the following steps:

- 1) Cross-compile the input program to a RISC-V ELF binary using `riscv64-linux-gnu-gcc`.
- 2) Extract the `.text` section from the binary and compute its SHA-256 hash using Python.
- 3) Compress the hash into four constants by XOR-ing its 64-bit chunks, and define these constants as C macros.
- 4) Recompile the program with these macros injected into a template verifier file, producing a final binary called `patched_main.elf`.
- 5) Optionally, flip a bit at a specified virtual memory address (VMA) using a custom Python script to generate `altered_main.elf`.

Finally, both the original and altered binaries are executed in QEMU using the following command:

```
qemu-riscv64 -L /usr/riscv64-linux-gnu <binary>
```

3.2 Techniques

We attempt 2 mitigation approaches: (1) Modifying TinyC compiler to emit additional instructions that detect and fault in the event of a bitflip and (2) **SHIELD**: Modifying the linking process to ensure user binaries contain additional protection instructions. We describe both approaches below.

3.2.1 Approach 1: Modify TinyC Compiler

Implications of Load/Store Architecture: RISC-V is a **load-store architecture**. This means that operands must exist in registers and ALU operations cannot occur on raw memory addresses. Unlike x86, a **register-memory architecture** that would require additional protections against bitflips in raw memory addresses, RISC-V only requires protection of registers, immediate values, and opcodes and their specifiers (minor opcodes in `funct3/funct7`). We classify these protections into **Stateful** and **Stateless** protections that indicate whether we can use state to help mitigate random bitflips.

Stateful Protections: The following outlines our approaches for protecting bitflips in registers. In these approaches, we leverage the fact that registers contain state that can be saved and verified. For the following 3 mitigations, we define an **Prologue** that (1) Calculates registers 1 Hamming Distance away ('neighbor registers') from the target register and (2) Saves original neighbor register values on the stack. We also define an **Epilogue** that (1) Restores original neighbor registers from the stack. Our approaches are detailed below and summarized in Figure 2.

- **Protect Single Register Dereferences:**

- **Prologue**
- Zero neighbor registers. Thus neighbor registers are guaranteed to fault if de-referenced.
- Perform the original instruction.
- **Epilogue**

- **Protect Single Register Writes:** `rd` – Since writes are blind, we need to add additional checks to ensure the results did not get written into the wrong register.

- **Prologue**
- Zero neighbor registers.
- Perform the original instruction.
- Verify neighbor registers of `rd` are still 0. If not zero (and thus written to), then fault.
- **Epilogue**

- **Protect Single Register Reads:**

- **Prologue**
- Copy the target register value from the original instruction into the neighbor registers. This means any bitflip in the read register produces no observable effect. Zeroing the register will not help, since we are not dereferencing, but instead using the raw value.
- **Epilogue**

Design Limitations of Approach 1: We find this approach, though a strong initial effort, is not very robust. In particular, this design has the following limitations:

- **Stateless protections:** In the previous two protections we were able to use the program itself to check the side effects of the code on the program state to determine

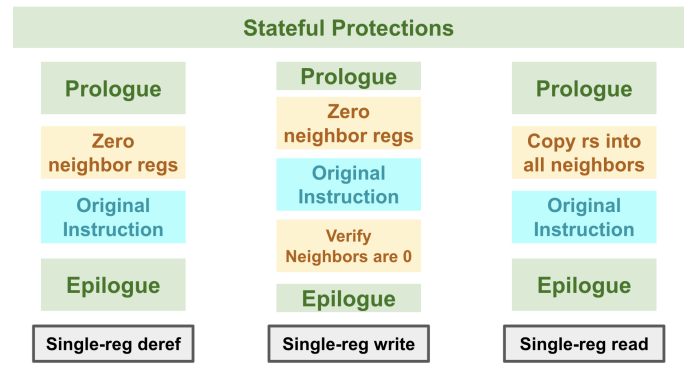


Fig. 2: Stateful Protections

if the code changed. However, when we think about opcodes and immediate, these are not part of the program state, but rather part of the program itself. It's impossible to prevent a `bne` flip to a `beq` using state alone; this is why the SWIFT paper (SEU model) used signatures to combat control flow checks.

- **Handling Multiple Registers in the Same Instruction:** No RISC-V instruction performs multiple writes at once, but there are many instructions that perform multiple reads/dereferences. We need additional logic to ensure our bitflip mitigations are not invalidating intended program control flow. Specifically, say our instruction we wish to protect is `add rd, rs1, rs2`. If `rd`, `rs1`, `rs2` are more than 1-hamming distance away from each other, then our current mitigation strategies can safely apply. However, if they are mutual neighbors, then we would need to ensure our mitigation strategy does not break control flow. For example, as part of our existing mitigations, `rd` may set `rs1` to 0, which can cause an incorrect result to be calculated.
- **Recursive Bitflips** – By adding protection instructions, we introduce additional areas where bitflips can occur and break our protections themselves. Rather than protecting our protection code, which is endlessly recursive, we would need additional guarantees that ensure a bitflip in our protection either faults or preserves intended control flow.

Though we initially tried to create mechanisms to counter these limitations (including restricting usable registers to those that are more than 1 HD apart), we thought it would be more interesting to explore a hashing based approach. We found our hashing based approach proved more robust than this attempt.

3.2.2 Approach 2: SHIELD

Secure Hashing Integrity Execution Linked Defense (**SHIELD**) is motivated by hashing a given program at compile time and dynamically verifying the hash matches at runtime. This yields the following design decisions:

- **How to efficiently hash a given program?** We leverage OpenSSL's SHA-256 RISC-V implementation.
- **How to hash a currently running program?** Only hashes only the text section of the file (code specific to the user program). Either use `/proc/self/exe`

file or the `__text_start` and `__text_end` symbols exported by our linker script.

- **How to check hashes align in an immune to bitflips?**
We need to be intentional about our sequence of hash checking and fault instructions to ensure they are immune to bitflips.

Figure 3 depicts the overview of our approach. On a high level, SHIELD adds a constructor that runs before the user program in a separate section of the binary. This constructor checks the user program’s SHA-256 hash at runtime matches the hash computed at compile time. We leverage OpenSSL to calculate SHA-256 hashes of given programs. When a user compiles with our build process and links with our carefully written `layout.ld` (shown in Listing 1), `template_verify.c` file (compiled into `verify.o`) will add a `.verify` section to the final ELF. This `.verify` section will contain (1) placeholders to later embed the compile time hash and (2) assembly that verifies the compile time hash matches the runtime hash (faulting in the event of a mismatch). This resultant `templated_main.elf` then gets passed into our custom `build.sh` file that calculates the compile time hash of the `.text` section of `templated_main.elf` and embeds the compile time hash into the final ELF `shielded.elf`. We note that we ONLY hash the `.text` section (that should contain user code based on the layout shown in Listing 1). Since we do not hash the `.verify` section, we carefully reason about why any bitflips in the constructor (protection code) will still correctly fault. This justification is included in Section 4.2.

```
SECTIONS {
  . = 0x400000;
  .text : {
    __text_start = .;
    *(.text)
    *(.text.*)
    __text_end = .;
    # Ensure we dont hash the constructor
    PROVIDE(__global_pointer$ = __text_end + 0x1000);
  }
  . = 0x800000;
  .verify : {
    *(.verify)
  }
  .init_array : { KEEP(*(.(init_array)) ) }
  .rodata : { *(.rodata*) }
  .data : { *(.data*) }
  .bss : { *(.bss*) *(COMMON) }
  /DISCARD/ : { *(.comment) *(.note*) }
}
```

Listing 1: `layout.ld`

We find SHIELD is free from the limitations of the previous approach. Specifically, since we do not change user-given code, we do not introduce data hazards with our protection instructions as our verification runs in a **constructor**, before `main`. Additionally, as detailed in section 4.2, our protection instructions are chosen very carefully, such that a bitflip within this section will correctly yield a fault.

Design Limitations of Approach 2 (SHIELD): Our design relies on two key assumptions that are realistic under

a Rowhammer threat model:

- 1) **Constructor integrity:** We assume the adversary cannot redirect the program’s entry point or disable our `init_array` constructor. A bitflip that hijacks the very first instruction pointer could bypass the hash check entirely.
- 2) **Pre-magic protection:** We assume no bitflips occur in the “magic” jump sequence immediately following our PC-plus-hash computation. That sequence is crafted so that any single-bit corruption either still transfers control into the user code or triggers a trap.

Under these assumptions, SHIELD has the following remaining limitations:

- **Unhashed data dependencies:** We only include the `.text` section in our hash. If control flow depends on values in `.data`—for example, global flags or table indices—a bitflip there goes undetected.
- **Earlier constructors:** If another constructor in the same `init_array` runs before ours and is itself corruptible, it could jump over or disable our check.
- **Hash compression collisions:** We compress the 256-bit SHA-256 digest to 64 bits via XOR. In the random-oracle model the chance of a collision is negligible but nonzero—an extraordinarily unlucky flip could yield a matching 64-bit value despite tampering.

We plan to address these gaps in future work by extending coverage to data sections, hardening earlier constructors, and investigating stronger collision-resistant compression schemes.

4 IMPLEMENTATION

4.1 Approach 1: Modify Tiny C

To implement our Stateful protections, we created API functions to protect single register dereferences, writes, and reads. Specifically, we modified `tinycc/riscv64-asm.c` and created new C functions to perform these protections. These C functions build upon the TinyC data structures for OP codes and functions to emit assembly instructions.

- `get_bitflip_registers` finds all registers 1 HD away from a given register
- `save_nearby_registers` performs the **prologue** and saves neighbor registers of a given register onto the stack. Sets neighbor registers to 0.
- `restore_nearby_registers` performs the **epilogue** and restores neighbor registers of a given register from the stack.
- `verify_nearby_rd_unchanged` assists in the protection of **single register writes**. Initially, when we tried a similar strategy to the protection for single-register dereferences, we did not correctly fault, since the RISC-V architecture does not read/dereference the register before writing. In other words, writes are blind, so our previous strategy of saving, zeroing, and restoring registers is not a sufficient protection against bitflips in `rd`. Thus, we needed a stronger mitigation that checks and faults if a neighbor register was written. To do this, we check the nearby registers remain 0 (indicating none was written into).

SHIELD Overview

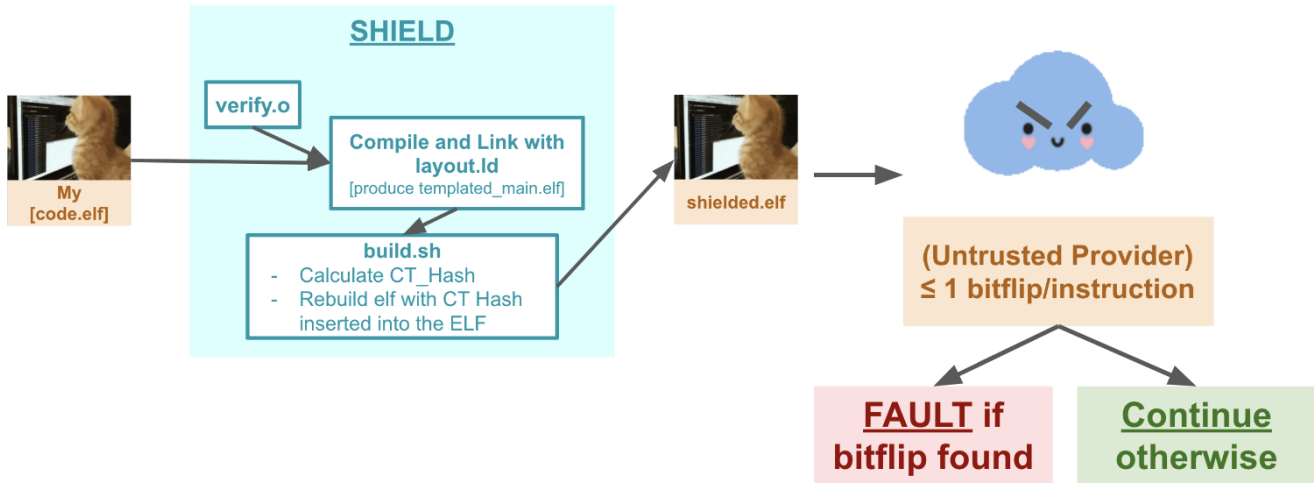


Fig. 3: Overview of our approach: SHIELD

- `copy_nearby_registers` performs the protection strategy for **single register reads**. Specifically it copies the value from a given register into all its neighbors. This ensures the intended value is always read.

Since we initially used `lw` as a case study, we added these protections around the `lw` instruction. Ultimately, due to the design limitations of this approach, we did not continue building out this implementation to serve more general cases.

4.2 Approach 2: SHIELD

4.2.1 Constructing a Simplified Proof of Concept

To outline our pipeline, we first assumed that the attacker could not bitflip our protection instructions. We defined `layout.ld` to have a `text` section followed by a `verify` section. We drafted `template_verifier.c` to define the `verify` constructor and use OpenSSL to hash the `.text` region (thereby calculating the runtime hash). We then use a simple `memcmp` to compare the Compile Time has (which we defined as a global variable `CT_HASH`) with the runtime hash and exit in the event of a mismatch. We compile and link the user program against our `template_verifier.c` using our `layout.ld` to produce an ELF.

We then leveraged the Python package `Library to Instrument Executable Formats (LIEF)` to parse the ELF and calculate a hash for the `.text` section (thereby producing the compile time hash). We then overwrite the `.data` section of the compiled ELF to store the calculated compile time hash in `CT_HASH`. This simple proof of concept was successful at checking compile time and runtime hashes were equivalent and exiting in the event of a mismatch.

Under the hood, an `if` statement on the result of `memcmp` calls the `beq` RISC-V instruction. Unfortunately, `beq` is 1 bitflip away from `bneq`. This means if an attacker could perform bitflips in the `.verify` section, then they could

easily flip our `beq` into a `bneq` and jump over our trapping instruction. Thus, we need to add more carefully crafted instructions that guarantee a fault if a bitflip occurs in the `.verify` section.

4.2.2 Protection Instructions Immune to Bitflips

To extend our proof of concept, we remove the assumption that `.verify` is immune to bitflips. Thus, we needed to more carefully design a sequence of instructions that would prevent attackers from reaching `main` in the event of a bitflip either in `.text` or in `.verify`. We show the overview of our design in Listing 2.

Though not shown in Listing 2, we first use the OpenSSL library to calculate a runtime hash of the `.text` section and store the result. We then obtain the program counter (PC) using `auipc` and add the compile-time hash to it. Rather than extracting the compile-time hash from the `.data` section, we encode the hash directly into the protection instructions. Section 4.2.3 describes how the compile-time hash is embedded into the protection logic. By line 9 of Listing 2, `x20` stores the sum of the PC and a 64-bit representation of the compile-time hash (Section 4.2.3 explains how we compress the SHA-256 hash into 64 bits).

Next, we move the runtime hash computed earlier into `x25` and subtract it from the value in `x20`. If no bitflips have occurred, then the compile-time hash equals the runtime hash, and `x20` stores the original PC. Otherwise, `x20` holds a random address in the 2^{64} address space due to the random oracle model of SHA-256. We assume the user program is memory-mapped into a limited portion of this space, so any address outside the valid range will cause a page fault. The chance of jumping to a valid address is negligible—for example, if the code size is 1 GB, the probability is approximately $2^{30}/2^{64} \approx 5.82 \times 10^{-9}\%$. In this way, we effectively create a conditional jump using only an unconditional register jump.

We then add an offset to skip a trap instruction. After line 18, `x20` should contain the address of the user program if no bitflips occurred or should contain a faulting address. Afterwards, we do an unconditional jump using `jalr` to skip the trapping instruction.

```

1  # x20 has PC
2  auipc x20, 0x0
3
4  # x23 = Compile Time Hash (omitted here for brevity)
5  # Use bit-shifting and math to load this into x23.
6  # Detailed in Section 4.2.3
7
8  # x20 = CT Hash + PC
9  add   x20, x20, x23
10
11 # x25 = RT Hash*/
12 mv   x25, %0
13
14 # x20 = (PC+CT Hash) { RT
15 sub   x20, x20, x25
16
17 # Calculate main's offset from auipc
18 addi  x20, x20, 0x50
19
20 # MAGIC Instruction!
21 jalr  x0, x20, 0
22
23 # TRAP
24 .word 0xFFFFFFFF

```

Listing 2: Overview of protection instructions

To choose protection instructions that remain safe even if a single bit flips, we created a web visualization of the RISC-V architecture using open source translations from hex to RISC-V. Our visualization (see: Figure 7) also shows, for any 32-bit RISC-V encoding, all 32 neighbors obtained by flipping one bit. This neighbor graph lets us immediately see which alternate opcodes, registers, and immediates would arise under a one-bit fault.

If we naïvely used a single `beq` or `bne` to branch to a trap on mismatch, the neighbor graph shows they are directly connected—flipping the one opcode bit that distinguishes `beq` (`funct3=000`) from `bne` (`funct3=001`) simply inverts the branch’s sense, allowing an attacker to skip the trap. Likewise, a single `lw rd, 0(x20)` instruction has neighbors that change into an `add`—which does not ensure a fault. By contrast, `jalr x0, x20, 0` uses a register-indirect unconditional jump. Its neighbor graph (Figure 7) shows that every single-bit variant either:

- Still jumps to invalid code
- Essentially becomes a NOP, allowing the control flow to run our trap: all 1 instruction is a machine translation error on line 24.

In practice we select the exact encoding `jalr x0, x20, 0` for the “magic” jump. Every one-bit neighbour of this instruction either traps immediately (illegal opcode, unmapped register, or mis-aligned target) or still performs a `jalr` to an address we control, so a single-bit Rowhammer fault in the protection stub cannot be coerced into useful

control-flow hijacking. Before the jump executes we compute the runtime hash of `.text` with OpenSSL’s SHA-256 and XOR-fold the 256-bit digest into a 64-bit value; that folding is the only arithmetic performed on the digest and is branch-free. More generally, in the RISC-V ISA a single-bit fault can only turn one conditional branch into another branch—no other instruction class is one bit away—so as long as our prelude contains *no* branches, every fault must fall through to the final `jalr`. (We did not prove that OpenSSL’s SHA-256 implementation is branch-free, so the hash function itself remains outside our trusted computing base; in principle one could replace it with a fully unrolled, branchless version.) Finally, although an “arbitrary read” gadget might in theory scan the constructor to recover the compile-time hash embedded in its immediates, those bits are scattered across multiple sign-extended `lui/li/ori` fragments, making reconstruction infeasible without exhaustive search; we therefore consider this threat negligible.

4.2.3 Design Challenges and Mitigations

4.2.3.1 Compressing the 256-bit digest to fit in 64-bit registers: The run-time constructor can only manipulate 64-bit values, so we reduce the full SHA-256 output by *XOR-folding* the four 64-bit words of the digest into a single 64-bit checksum. While-folding clearly weakens the collision guarantee, the resulting 64-bit value still gives 2^{-64} collision probability in the random-oracle model—ample for detecting a single accidental or adversarial bit-flip.³

4.2.3.2 Why `.verify` must not be hashed: At first we attempted to hash the entire image, including the constructor itself. That failed: on the first link pass, the `CT_HASH` placeholder contains zeroes; after we compute the digest and patch the binary, the contents of `.verify` change, invalidating the very digest we just wrote. The remedy is to:

- 1) Hash *only* `.text`, whose contents are immutable after linking occurs.
- 2) Place `.verify` *after* `.text` (Listing 1) so that control flow must pass through the constructor before any user code can run. We also pin the RISC-V global-pointer (`gp`) to a fixed address *before* any later sections are laid out; if we let the linker choose the default location inside `.sdata`, the value of `gp` (and every `gp`-relative immediate already baked into `.text`) would shift whenever we append or resize `.verify`, invalidating the compile-time hash.
- 3) Export `__text_start` and `__text_end` from the linker script so the constructor can hash its precise bounds without relying on file offsets or use introspect via its `/proc/self/exe` ELF.

4.2.3.3 Encoding the hash with signed immediates: RISC-V immediates are *signed* (20 bits in `lui`, 12 bits in `addi/li`). If the high bit of an immediate is 1, the assembler sign-extends it, corrupting the value. We therefore split each 32-bit half of the checksum into “high-20” & “low-12” chunks, then split each chunk again so that no field feeds a sign bit:

3. Using a native 64-bit hash would be better; we chose XOR-folding to avoid introducing another dependency into our minimal RISC-V userspace.

```

li    x23, TOP_32L_11      # 11 safe bits
slli x23, x23, 1
ori   x23, x23, TOP_32L_0 # add the 1 LSB
slli x23, x23, 32         # move to bits 32..43
add   x20, x20, x23

lui   x23, TOP_32_19      # 19 safe bits -> bits 31..12
srai  x23, x23, 11        # undo LUI's <<12 & sign
ori   x23, x23, TOP_32_1  # add the 1 LSB
slli x23, x23, 44         # move to bits 44..63
add   x20, x20, x23

```

By construction every immediate fed to `lui` or `li/addi` has its sign bit clear, so the assembler never corrupts the embedded hash.

5 EVALUATION

5.1 Testing Program Response to Bitflips

5.1.1 *bitflip.py*

To evaluate the efficacy of our solutions, we created a python script that allows us to manually perform a bitflip at a specified byte address and offset. This script extracts the text section of a given ELF file, converts the user-given parameters into a byte offset and flips the bit. The result is written to an output file specified by the user.

5.1.2 *Sanity Tests*

Overall, to check whether our protection mechanisms are successful, we ensure (1) program behavior remains unchanged when there is no bitflip and (2) the program faults if there IS a bitflip present in the program.

Approach 1 succeeds at mitigating exactly 1 bitflip in the program, given the bitflip occurs at a register code of a known target instruction. Figure 8 demonstrates how we check this mitigation strategy works for the case of **single register writes**. Unfortunately, due to the recursive issue (where protection instructions lack protection themselves), this approach is not invulnerable to multiple bitflips or bitflips attacking opcodes/immediates. Additionally, this solution is not robust to when the user does not know where the bitflips occur, since we only use our APIs in specific instructions. Due to the inherent design limitations of this approach, we did not try to continue scaling it to more diverse scenarios.

For Approach 2, to validate that our constructor really converts *any* single bit-flip into a safe crash, we run the helper script `run.sh` in two modes:

- 1) **Baseline.** Compile and execute the protected binary (`patched_main.elf`) with no faults injected.
- 2) **Fault injection.** Flip one bit inside the `.text` section (VMA `0x400100` in this example), save the result as `altered_main.elf`, and execute it under the same QEMU environment.

Figure 4 summarises the session. With no injected fault the constructor computes the run-time SHA-256 digest, finds it identical to the embedded compile-time checksum, and performs the “magic” `jalr` that skips the trap sled and reaches `main`, which correctly prints “*Hello! You cannot change my code! Hahaha!*”

After flipping a single bit in `.text`, the folded digest no longer matches; the same `jalr x0, x20, 0` therefore targets an unmapped address and QEMU immediately raises a *Segmentation fault*. Because every 1 HD neighbour of that `jalr`

```

> ./run.sh tests/simple.c
[5] Flipping bit in patched_main.elf...
✓ Flipped bit 0 at VMA 0x400100 (file offset 0x1100)
Saved to ./altered_main.elf
[✓] Bit flip complete: altered_main.elf
[✓] Executing patched_main.elf with QEMU...
Hello!
You cannot change my code!
Hahaha!
[✓] Executing altered_main.elf with QEMU...
./run.sh: line 74: 497701 Segmentation fault (core dumped)
qemu-riscv64 -L /usr/riscv64-linux-gnu -E LD_LIBRARY_PATH=$LD_LI
BRARY_PATH ./altered_main.elf

```

Fig. 4: SHIELD run—one injected bit-flip. The unmodified binary prints its normal output; the tampered binary crashes in the constructor with a segmentation fault, *before* any user code executes.

encoding is either illegal or still jumps outside the code segment (Figure 7), an adversary cannot steer execution past the trap with only a single Rowhammer-style fault.

We repeated the experiment with the bit-flip placed inside the `.verify` section itself (i.e. inside the constructor). Any corruption of the “magic” sequence likewise ends in a trap or an off-segment jump, confirming that the protection stub is *self-protecting* under the one-bit threat model.

5.2 Performance Overhead of SHIELD

Due to the design limitations of Approach 1, we focus our evaluation on our final design: Approach 2 (SHIELD). To benchmark this solution, we created a benchmarking bash script that ran the original ELF and the SHIELD-protected ELF for ($N = 5$) rounds and saved the end-to-end execution time to a CSV. These measurement includes setup and cleanup of the running binary, since we start the timer before calling the QEMU runtime and end the timer after QEMU runtime finishes.

We first created a simple test suite. As shown in Figure 5, we find that in this simple suite, SHIELD adds around 60-70 ms of overhead.

- `simple.c` just performs print statements
- `network.c` performs an HTTP request. This is a more realistic benchmark.

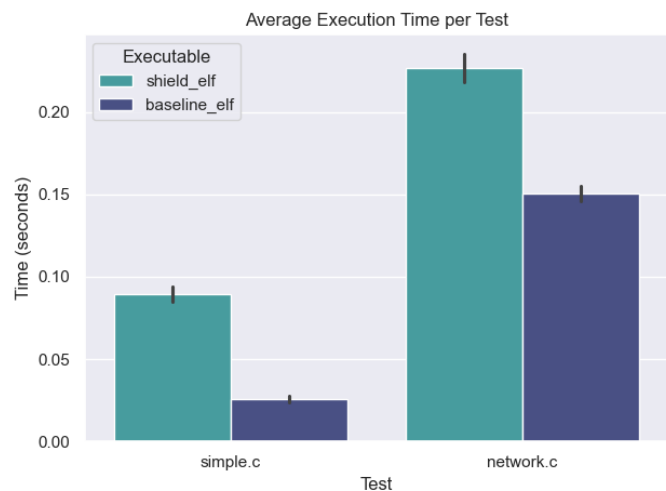


Fig. 5: Simple Benchmark Suite.

Since our solution does rely on hashing the currently running binary, we wanted to explore whether the size of the binary would impact overhead of SHIELD. To create this benchmark, we created a python script that generated C files that called M kilobytes of NOP instructions. Though additional NOP instructions should not significantly impact user program runtime, they do add more lines to input that needs to get read and hashed at runtime. Thus, we initially thought that as binary size increased, the gap between `shield_elf` and `baseline_elf` would widen since SHA-256 processes inputs in 512-bit (64 byte) chunks.

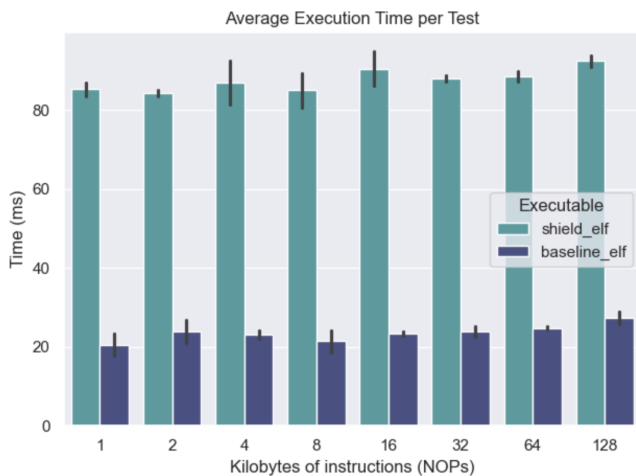


Fig. 6: Benchmark Suite of Varying bytes of NOPs

As shown in Figure 6, our hypothesis was not fully validated. These could be because our tested program inputs were perhaps too small to stress the very efficient SHA-256 algorithm. Thus we find our SHIELD solution seems to have a relatively small overhead while offering strong protection guarantees.

6 CONCLUSION AND FUTURE WORK

Contributions

This project presents a defense against Rowhammer-style attacks in the RISC-V architecture, assuming a strong adversary model: the attacker may induce *at most one bitflip per 32-bit instruction*. In contrast to much of the prior work that targets single-event upsets, our model assumes the adversary has an arbitrary Rowhammer gadget at runtime. Under this model, we make the following contributions:

- We defined a **stronger adversary model** where the attacker can induce a single bitflip in each instruction—one not widely explored in prior work.
- We evaluated **Approach 1**: modifying the Tiny C Compiler to insert fault-detecting register logic directly into the emitted RISC-V code. Though ultimately unscalable due to recursive protection and data hazards, this approach provided important insights into ISA-level fault detection.
- We designed and implemented **Approach 2 (SHIELD)**: a robust hashing-based mitigation scheme that uses a constructor to compute and compare compile-time and runtime hashes of the program’s `.text` section,

thereby detecting bit-level tampering with high probability.

- We created a **RISC-V bitflip visualization tool** that illustrates the one-Hamming-distance neighbors of any instruction. This helped us identify `jalr x0, x20, 0` as a fault-tolerant “magic jump” with desirable safety properties.
- We developed a complete **build system** that compiles, instruments, hashes, and patches ELF binaries to include our verification code and compressed hash.
- We conducted **benchmarking and empirical validation** of our techniques using QEMU, and showed that SHIELD offers reliable protection with low runtime overhead.

Future Work and Extensions

Several promising extensions could further improve the robustness, speed, and generality of our defense:

- **Granular Hashing**: In our current design, SHIELD performs a one-time hash verification at startup, assuming the adversary flips bits only before program execution begins. However, our defense can naturally extend to a stronger adversary model where bitflips occur *at any point during runtime*. To support this, we propose introducing intermediate hash checks at critical control-flow boundaries (e.g., function calls, loops, or system calls). Conceptually, this approach builds an inductive invariant: if the initial part of the program starts in a verified, unmodified state, and before every jump to another code region it revalidates the current instruction stream, then any post-startup bitflip will be detected before it can influence control flow. It no longer would be instrumented using a constructor, we could just directly insert a call to a new verify instruction instance and the hash could be in the data section as we now have guaranteed this invocation has no bitflips. We can estimate this adds modest runtime overhead, but tightens our security guarantees against transient, time-varying fault injection and opens the door to formally proving full execution-time integrity under our bitflip model.
- **ISA-Level Fixes**: Design a new RISC-V `branch` instruction that is not one bit away from other conditional branches. This would provide deterministic protection against control-flow hijacking and eliminate the need for our probabilistic `jalr`-based “magic jump.”
- **Hardware Hashing**: Leverage the RISC-V crypto extension for faster runtime hashing. While we currently rely on OpenSSL in userspace and QEMU emulation, using hardware-accelerated SHA-2 would significantly reduce verification time—though this requires testing on a real RISC-V CPU and also ensuring these instructions (already implemented in QEMU) don’t bitflip to branches.
- **Improved Evaluation**: Move from QEMU-based benchmarks to real RISC-V hardware to obtain accurate runtime measurements and rule out emulator-induced noise.
- **Hash Improvements**: Replace our XOR-compressed SHA-256 digest with a more robust compression

method or use a native 64-bit cryptographic hash function (e.g., BLAKE2s or SHA-512/64) to reduce collision probability without relying on compression.

Together, these directions can strengthen defenses against fault injection, reduce performance overhead, and bring this style of runtime integrity checking closer to practical deployment on real RISC-V systems.

REFERENCES

- [1] Thierno Barry, Damien Couroussé, and Bruno Robisson. Compilation of a countermeasure against instruction-skip fault attacks. In *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*, pages 1–6, 2016.
- [2] Fabrice Bellard. Tiny c compiler - c scripting everywhere - the smallest ansi c compiler. <https://github.com/TinyCC/tinycc?tab=readme-ov-file>.
- [3] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *Proceedings of the 28th USENIX Conference on Security Symposium, SEC'19*, page 249–266, USA, 2019. USENIX Association.
- [4] Zhi Chen, Junjie Shen, Alex Nicolau, Alex Veidenbaum, Nahid Farhady Ghalaty, and Rosario Cammarota. Camfas: A compiler approach to mitigate fault attacks via enhanced simdization. In *2017 Workshop on Fault Diagnosis and Tolerance in Cryptography (FDTC)*, pages 57–64, 2017.
- [5] Daniel Gruss, Moritz Lipp, Michael Schwarz, Daniel Genkin, Jonas Juffinger, Sioli O’Connell, Wolfgang Schoechl, and Yuval Yarom. Another flip in the wall of rowhammer defenses. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 245–261, 2018.
- [6] Radhesh Krishnan Konoth, Marco Oliverio, Andrei Tatar, Dennis Andriese, Herbert Bos, Cristiano Giuffrida, and Kaveh Razavi. {ZebRAM}: Comprehensive and compatible software protection against rowhammer attacks. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 697–710, 2018.
- [7] QEMU. Qemu: A generic and open source machine emulator and virtualizer. <https://www.qemu.org>.
- [8] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. Swift: software implemented fault tolerance. In *International Symposium on Code Generation and Optimization*, pages 243–254, 2005.
- [9] Mark Seaborn and Thomas Dullien. Exploiting the dram rowhammer bug to gain kernel privileges. *Black Hat*, 15(71):2, 2015.
- [10] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. *SIGOPS Oper. Syst. Rev.*, 27(5):203–216, December 1993.
- [11] Zachary Yedidia. Lightweight fault isolation: Practical, efficient, and secure software sandboxing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS '24*, page 649–665, New York, NY, USA, 2024. Association for Computing Machinery.



Fig. 7: We generated a visualization that shows all instructions 1 bitflip away from a given instruction. This figure demonstrates the neighbors of `jalr x0, 0(x20)`

```
prachi@shankara:~/cs380s/rowhammer-patch$ ../../tinycc/riscv64-tcc play/syllabus.S -o play/syllabus.out
bash: ../../tinycc/riscv64-tcc: No such file or directory
prachi@shankara:~/cs380s/rowhammer-patch$ ./tinycc/riscv64-tcc -L/usr/riscv64-linux-gnu/lib play/syllabus.S -o play/syllabus_bothregs.out
prachi@shankara:~/cs380s/rowhammer-patch$ riscv64-linux-gnu-objdump -S -M no-aliases play/syllabus_bothregs.out |nvm
prachi@shankara:~/cs380s/rowhammer-patch$ python3 play/bitflip.py play/syllabus_bothregs.out play/syllabus_bothregs_flip.out 0x119c9 1
✔ Successfully flipped bit 1 at VMA 0x119c9 (File Offset: 0x9c9) in play/syllabus_bothregs.out
📁 New binary saved as: play/syllabus_bothregs_flip.out
prachi@shankara:~/cs380s/rowhammer-patch$ riscv64-linux-gnu-objdump -S -M no-aliases play/syllabus_bothregs_flip.out |nvm
prachi@shankara:~/cs380s/rowhammer-patch$ qemu-riscv64 -L /usr/riscv64-linux-gnu play/syllabus_bothregs.out
prachi@shankara:~/cs380s/rowhammer-patch$ qemu-riscv64 -L /usr/riscv64-linux-gnu play/syllabus_bothregs_flip.out
Segmentation fault (core dumped)
prachi@shankara:~/cs380s/rowhammer-patch$
```

Fig. 8: As an example of our workflow, we test our protection of **single register writes**. Line 2 builds `tinycc` with our changes. Line 3 allows us to find the target byte we want to flip. Line 4 flips the bit. Line 5 allows us to verify the correct bit flipped. Line 6 shows original code runs as intended and Line 7 shows flipped code properly faults.