HFI in RISC-V

Dinesh Keserla (dk27833)
University of Texas at Austin
dineshkeserla@utexas.edu

Dev Patel (dbp748)
University of Texas at Austin
devpatel199@utexas.edu

Joseph Stanley (jps4455)
University of Texas at Austin
stanley.p.joseph@gmail.com

1. Project Goals

Hardware-assisted fault isolation (HFI) on RISC-V aims to address two critical needs in modern computing systems: preventing security bugs by isolating untrusted or risky code, and better understanding (and improving) performance overheads that arise from current software-based or OS-based sandboxing mechanisms. Fine-grained sandboxing is essential here as a single defective or malicious component can be contained before it compromises the rest of the application or operating system. By offering a minimal set of architectural extensions for in-process isolation, HFI empowers developers to sandbox components (e.g., libraries, plugins, user-defined functions) without resorting to heavyweight processes or virtual machines. Ultimately, the project's goal is to demonstrate a secure yet performant solution that seamlessly integrates into existing RISC-V designs.

Several recent high-profile vulnerabilities—such as Heartbleed (CVE-2014-0160) and Spectre (CVE-2017-5753, CVE-2017-5715)—underscore the need for stronger isolation between mutually distrusting code modules. In each of these incidents, the lack of strong intra-process boundaries allows an attacker to read or corrupt sensitive data; a robust sandbox would have limited the impact of the attack to the faulty module itself and preserved overall system integrity. Compared with classic software-based fault isolation (SFI), a hardware sandbox not only reduces runtime overhead but also eliminates whole classes of security issues that have historically plagued SFI implementations such as side-channel attacks and out-of-bounds errors. System architects and software developers alike have turned to hardware features for efficient compartmentalization. In this spirit, our HFI design for RISC-V builds on prior work that introduced HFI for x86-64 architectures, enabling fast transitions into and out of "sandbox" contexts while enforcing strong data and control-flow checks.

2. Background

2.1. Software-based Fault Isolation

Modern applications consistently rely on a large number of various libraries for key functionality, however, this leads to a problem as bugs within these libraries can become a vector of attack for the entire application. To mitigate this, it is critical to contain the scope of what each library can access or affect. Traditionally, general isolation was considered at the granularity of processes, containers, and virtual machines, which do achieve the functionality of isolation of libraries through hardware-level protection via the MMU and user-kernel privileges. Although these isolation methods are functional, they have significant performance overhead from the context switching and memory management. Due to the fact that the executing libraries are quite small, it motivates isolation that is fine-grained and lightweight.

These issues are remedied by Software-based Fault Isolation (SFI), which performs isolation in userspace via compiler instrumentation. SFI creates a sandbox, which is a restricted execution environment where untrusted code is contained in regards to both memory safety and control-flow integrity. Memory accesses are restricted via bounds checks or address masking, while control-flow integrity is maintained by constraining jumps to valid, pre-approved destinations. Additionally, to prevent unauthorized interaction with the operating system, system calls from sandboxed code are redirected to a trusted runtime, which will validate and mediate them before execution.

This process of fine-grained isolation within a single address space of the application is referred as *in-process isolation.* Recent SFI approaches, including WebAssembly's use of guard regions and JIT-based SFI schemes, have shown that in-process isolation is viable but remains limited by performance bottlenecks and compatibility issues. For instance, large FaaS (Function-as-a-Service) platforms and browser engines isolate tens of thousands of small code modules, resulting in high context-switch overheads or inordinate virtual-address consumption. More explicitly, SFI systems face the following limitations:

1) Performance overhead: SFI implementations can impose significant runtime overhead with over 40% in practical workloads. This stems from the cost of inserting and executing additional instructions for validating memory loads and stores, control-flow validation, and system call interposition.

2) Limited scalability: SFI schemes rely on allocating large, sparsely-mapped virtual memory regions with guard pages (8 GiB per sandbox in

WebAssembly) in order to validate memory with the much more performant masking as opposed to base and bound checks. This consumes vast amounts of virtual address space, making it impractical to host many concurrent sandboxes.

3) Poor compatibility: SFI requires code to be compiled or transformed in specific ways, which limits its compatibility with unmodified native binaries, low-level system libraries, and code that performs dynamic memory management or emits code at runtime.

4) Vulnerability to spectre attacks: SFI checks are instrumented as checks within the code and like any other code, modern processors may speculatively execute instructions past these checks, potentially leaking sensitive data through a side channel.

Despite these limitations, SFI is increasingly adopted due to its relatively low overhead compared to traditional isolation mechanisms.

2.2. HFI: Hardware-based Fault Isolation

To address the aforementioned limitations found in SFI, Hardware-assisted Fault Isolation (HFI) [1] was introduced. By consolidating and standardizing architectural support for these needs, HFI reduces reliance on kernel calls, shrinks the required memory footprint, and helps unify existing SFI-based runtimes under a lightweight, hardware-managed sandboxing mechanism.

Although hardware-based isolation solutions can be integrated at multiple levels, the approach mentioned in [1] focuses on tightening the security of low-level software runtimes responsible for loading, executing, and managing untrusted or semi-trusted code within the same process. Hardware-assisted Fault Isolation (HFI) relies on the following core mechanisms:

1) In-Process Isolation via Regions: The processor tracks a small set of metadata registers—each defining accessible addresses and permissions for code or data. When enabled, any out-of-bounds access or forbidden system call triggers an immediate trap, ensuring robust isolation. These hardware-enforced bounds checks happen in parallel with memory lookups.

2) Lightweight Sandbox Mode: A single hfi_enter instruction enables isolation, while hfi_exit disables it. Each sandbox can optionally redirect system calls, enabling complete interposition if the sandbox is untrusted. The overhead of enter/exit can be on the order of tens of cycles, much lower than classical kernel-based context switches.

3) User-Space Sandbox Configuration: By placing HFI instructions at the user level, ring transitions for memory setup are avoided. This design helps ephemeral tasks or microservices that create and destroy sandboxes frequently, as reconfiguring regions or resizing memory is a matter of writing to hardware registers, not calling into the OS.

4) Spectre Mitigations: HFI includes optional serialization on transition instructions, ensuring that speculative execution cannot bypass newly set bounds or leak privileged data outside the sandbox. Additional hardware checks prevent malicious code from polluting caches or branch predictors with secrets from other regions.

As demonstrated in prior x86 implementations [1], HFI's hardware cost is kept minimal, and the main processor pipelines remain largely untouched. This ensures that non-sandboxed code experiences negligible slowdowns, while sandboxed code reaps a significant performance advantage compared to purely software-based solutions.

2.3. QEMU

Quick Emulator (QEMU) is a free and open-source emulator for computer processors of varying architecture. QEMU is able to emulate various popular architectures such as x86, ARM, RISC-V, and more. The core functionality of emulating various CPU architectures is achieved through *dynamic binary translation*, in which guest machine instructions are converted to equivalent host machine instructions at runtime. This allows it to execute guest programs portably on host systems with different instruction set architectures.

QEMU leverages a Just-In-Time (JIT) compiler called Tiny Code Generator (TCG), which translates guest instructions into host instructions. This translation layer is extensible and can be used to add functionality to existing instructions or create new instructions. Additionally, the state inside the emulated architecture can be modified to add or remove registers and prepare for functional semantics of potential hardware modifications. Furthermore, since these modifications can be applied entirely in software, the need for RTL simulation or FPGA synthesis can be avoided, which would be significantly more complex and time-consuming.

While QEMU enables validation of functional correctness, it cannot capture microarchitectural vulnerabilities such as Spectre or Meltdown, which depend on speculative execution, out-of-order pipelines, and cache-based side channels. For this reason, QEMU is not suitable for evaluating timing attacks or microarchitectural side effects, but remains a powerful tool for verifying correctness at the architectural level. Moreover, verifying performance is not possible as QEMU lacks features for modeling cycles.

3. Security/Performance Currently

Software-only isolation approaches face significant security challenges. Even robust SFI frameworks, such as those

used in browser environments, face challenges from side-channel attacks, especially under cross-sandbox conditions where memory layout or branch predictors leak information.

In contrast, a hardware-assisted approach can unify bounds checking and control-flow constraints at the processor level, closing off out-of-bounds reads or writes before they ever reach caches or translation lookaside buffers. Furthermore, hardware-based mediation of system calls—critical for sandboxing unmodified native libraries—promises to eliminate entire classes of bypass vulnerabilities.

Beyond security, performance remains a central issue. Current software-based guard-page or compiler-instrumented approaches can incur 20-40% runtime overhead on CPU-intensive code, and ephemeral workloads (common in serverless contexts) may pay a high cost to instantiate and tear down per-sandbox memory. Typical solutions (e.g., mprotect, madvise) require system calls and TLB shootdowns that do not scale gracefully.

On RISC-V, these overheads become particularly relevant for microcontroller-class devices—where every cycle matters—and for large-scale data centers wanting to isolate thousands of workloads concurrently. As prior experiments with x86-based HFI show, carefully designed hardware instructions can reduce context-switch time to near that of a function call, while supporting user-space mechanisms for setting up sandbox memory regions. The challenge is thus to replicate and refine these ideas in a clean-slate RISC-V setting without compromising security or drastically altering the instruction set.

Equally important, HFI tackles concrete flaws that consistently slip past software guards. Software-only SFI cannot stop transient-execution attacks such as Spectre (CVE-2017-5753, CVE-2017-5715): an attacker can speculatively jump over the compiler-inserted bounds checks and leak data. Because HFI performs the range comparison inside the load/store pipeline and stalls the access until the check resolves, the classic Spectre gadgets are rendered unexploitable. The same in-pipeline enforcement also neutralises newer variants (e.g., Spectre v4 and Load Value Injection) without utilizing costly lfences throughout the code base.

At the same time, HFI preserves the ability to confine more "traditional" memory-safety bugs, such as out-of-bounds errors, that SFI already blocks—while doing so with better performance. Isolating OpenSSL in an HFI sandbox still prevents the Heartbleed over-read (CVE-2014-0160) from escaping into adjacent heaps, but avoids the 25–35% overhead that guard-page faults imposed in process-based SFI. Likewise, keeping vulnerable image decoders (e.g., the 2023 libwebp heap overflow) in an HFI region lets a browser switch into and out of the decoder in 80 ns—about the cost of a function call—rather than launching a heavyweight renderer process. In short, HFI simultaneously closes the speculative side-channels that SFI misses and slashes the overhead of the mitigations SFI already provides.

4. Approach

Our work brings HFI concepts to the RISC-V ecosystem by adapting the existing HFI design while fitting into RISC-V's existing privileged architecture. RISC-V presents both unique opportunities and challenges for HFI implementation. Its extensible ISA and clean-slate design allow for elegant integration of new instructions, while its simpler microarchitecture requires careful consideration of implementation details.

By leveraging RISC-V's custom opcode space, we can integrate the necessary HFI instructions without disrupting compatibility with existing software. Our implementation focuses on providing the essential HFI functionality while maintaining the performance benefits that make hardware-assisted isolation valuable.

Our approach aims to preserve the security guarantees demanded by modern sandboxing frameworks, while also reducing overheads that historically have hindered fine-grained in-process isolation in the RISC-V ecosystem.

Therefore, the RISC-V implementation in QEMU can be modified to add necessary HFI registers and instructions, allowing for validating HFI's ISA semantics and control-flow traps. Moreover, by implementing in QEMU, RISC-V binaries using the created HFI instructions can be verified for functional correctness. This allows the validation of memory safety enforcement, instruction semantics, and sandbox behavior across a variety of test workloads.

4.1. Custom Instruction Encoding in RISC-V

Before we consider custom instructions, we will first consider the key details of instruction encoding in RISC-V. For starters, RISC-V is a fixed instruction length architecture with two variant sizes 16-bit (compressed) and 32-bit. For our purposes, we will only need to consider the 32-bit instructions.

TABLE 1. RISC-V INSTRUCTION FORMAT ENCODINGS

31–25	24–20	19–15	14–12	11–7	6–0	Type
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type

Although Table 1 is not a complete representation of all the types of instructions in RISC-V, it will suffice for our implementation. Register-type (R-type) denotes instructions that function entirely on registers such as *add* and *or*. Immediate-type (I-type) are instructions that will require an immediate such as *lw* (load word) and *addi*. Store-type (S-type) are specified for specifically stores to memory where the immediate is split into two parts; an instruction of this variant is *sw* (store word). For our purposes R-type will be utilized for updating and retrieving state of the sandbox such as the size of the implicit regions, base and bound of explicit regions, and their associated permissions. I-type and S-type will be in regards to the loads and stores for the explicit regions with the h-prefixed instructions mentioned in Section 5.

In each of these instruction types, certain fields remain consistent, these are the first source register (rs1), the funct3 field, and the opcode. The funct3 and opcode fields are used together to distinguish between specific instructions. The opcode generally defines the broad category of the instruction, while funct3 provides finer-grained differentiation within that category.

TABLE 2. ENCODING OF RISC-V LOAD INSTRUCTIONS

imm	rs1	funct3	rd	opcode	Instruction
imm[11:0]	rs1	000	rd	0000011	lb
imm[11:0]	rs1	001	rd	0000011	lh
imm[11:0]	rs1	010	rd	0000011	lw
imm[11:0]	rs1	100	rd	0000011	lbu
imm[11:0]	rs1	101	rd	0000011	lhu
imm[11:0]	rs1	110	rd	0000011	lwu
imm[11:0]	rs1	011	rd	0000011	ld

For example, in Table 2, see that their opcode is the same indicating their broad category of load and the funct3 value is varied for the specialized instruction.

Currently, there are 4 opcodes that are available for extending the instruction set; these are custom-0 (0001011), custom-1 (0101011), custom-2 (1011011), and custom-3 (1111011). It is important to note that the opcodes custom-2 and custom-3 have been reserved for the open proposal for RV128, which will specify the RISC-V ISA for 128-bit machines. However, these remain available in RV32 and RV64. For simplicity, we will focus our efforts on extending RV64 to ensure no potential conflicts, but modifications should be considered to reduce the number of used opcodes and instructions to maximize compatibility.

In the composition of new instructions that interact with the underlying HFI state using the R-type instructions, we will use the custom-0 instruction. Additionally, if we return to Table 1, we can see that these R-type instructions also have a funct7 value which can be used to further increase the number of instructions with a fixed value for funct3 and opcode. Therefore, further consolidation can be performed for many of the R-type instructions. However, it should be mentioned that in creating instructions to expand functionality such as in Section 5, we did not use this as a focus, and determined this could be determined later. As a result, our chosen funct3 and funct7 values can be consolidated within many of our implemented instructions.

Furthermore, it can be seen that loads and stores have little room for expansion with the absence of a funct7 value, and for the explicit region problems arise, which are discussed in Section 6.

4.2. Extending QEMU for HFI: Instructions

Upon determining the instruction encoding, decoding the instruction requires modifiying the translation pipeline to recognize the instruction and translate it into the appropriate sequence of TCG operations for execution. This process is structured into three main components:

First, a decode rule must be added in the appropriate instruction format file (e.g., `insn32.decode`) to match the opcode and instruction fields. This rule identifies the custom instruction and extracts its parameters (such as `rd`, `rs1`, `imm`, etc.).

Second, a corresponding translation function must be defined in the instruction translation file (e.g., `trans_hfi.c.inc`). This function is written using QEMU's Tiny Code Generator (TCG) and is responsible for generating the intermediate representation that will be executed by the emulator.

Finally, a helper function may be defined (typically in C) to carry out the instruction's semantic behavior on the CPU. The translation function emits a call to this helper using TCG, and passes in the appropriate virtual CPU state and arguments. This separation allows the architectural logic to be written in C while still integrated into JIT-compiled TCG execution.

This process forms the standard method for implementing new instructions in QEMU, including those used in supporting architectural extensions like HFI.

4.3. Extending QEMU for HFI: State

Modifying the state of QEMU to support the state required for HFI involves extending the architectural state of the RISC-V CPU model to track region metadata and HFI status.

All state must be added to QEMU's `CPURISCVState` structure, which holds per-core architectural state. This enables extensions such as HFI to be modeled as a first-class architectural feature that can interact with instruction translation and memory access logic.

The added new state is zeroed out on initialization, but modifications are preserved across QEMU execution so that memory checks and instruction semantics can depend on it.

4.4. Region Enforcement Semantics

Region enforcement in QEMU for HFI can be achieved by emitting TCG-level checks that guard memory accesses when HFI is active. These checks validate that each memory access falls within the appropriate region bounds, either for implicit data/code regions or explicit ones depending on the instruction type.

For standard load and store instructions that operate under implicit region constraints, the necessary bounds checks must be inserted during translation. Specifically, TCG code should be emitted to verify that the computed memory address lies within a configured implicit region before allowing the memory operation to proceed. These checks must always be emitted unconditionally, because QEMU caches translated blocks. If the TCG check were omitted based on the runtime value of the HFI status, the translation block could be cached with a stale value, leading to incorrect execution that bypasses necessary HFI enforcement.

In contrast, HFI-prefixed load and store instructions perform region validation directly in their corresponding

C helper functions, which are invoked from TCG. These helpers are written to emit the necessary checks using standard C logic, and since the code is incorporated into TCG, the effect is functionally equivalent to always emitting the inline TCG code as with implcit regions.

Together, these approaches ensure that region enforcement is always active when required and that both implicit and explicit region checks are performed correctly and reliably across QEMU's translated execution model.

5. Implementation & Evaluation

5.1. Implementation - Intial Setup

Since we were implementing our RISC-V HFI extension in QEMU, we cloned the current open-source QEMU repository and began work there. We began by taking time to understand how QEMU works, the purpose of the various repositories, and gaining a better understanding of RISC-V. We worked with the assumption that the system being run on always supports HFI but it is an easy addition to add any necessary hardware checks to validate that assumption.

Our work then began by creating a basic internal register called *hfi_status*, which serves as a flag bit for verifying whether HFI can be toggled on or off via our newly implemented instructions. Through the use of QEMU logging we display the value of *hfi_status* and use it as a validator that the instructions have run. For our first instruction, we have *hfi_enter*, which sets *hfi_status* $= 1$ when executed. Additionally, we have *hfi_exit*, which sets *hfi_status* $= 0$. RISC-V contains unmapped opcodes that can be used for the addition of new instructions. In order to implement our instruction we have opted to use the free opcode 0001011. We are then able to differentiate between the two functions by modifying the funct3 value of the instructions such that *hfi_enter* has funct3 $= 000$ and *hfi_exit* has funct3 $= 001$. Adding the control and status register (CSR) *hfi_status* and implementing the *hfi_enter* and *hfi_exit* gave us basic functionality to enable and disable HFI.

5.2. Implementation - Implicit Regions

After testing whether we can toggle the *hfi_status* register properly, we moved on to implement implicit regions. As aforementioned, implicit regions are used for native, unmodified binaries and an important feature of HFI. The original HFI paper had four implicit data regions and two implicit code regions. In our implementation, we supported an arbitrary number of data and code regions which were defined by a global variable that could be adjusted as needed. We also tested using varying numbers of each region and ensured the HFI logic worked seamlessly with the defined number of regions. Implicit regions are defined by a base and a mask.

To store metadata about these regions, we created new structs *HFIImplicitDataRegion* and *HFIImplicitCodeRegion* which we stored in the CPU architecture state in arrays of length based on the number of regions we were supporting. To set these regions, we implemented the instruction *hfi_set_region_size* which took in a region number, base address, and mask and configured a new data or code region accordingly. Then, to set permissions for this region, we added a new instruction *hfi_set_region_permissions* which took a region number and a 64 bit integer encoding the permissions for that region. Since a single call to *hfi_set_region_permissions* will be used for a singular region (differentiated using the code region number), the permission encoding for each of the 3 types of region can also differ. Implicit data regions store the enabled bit at bit 7, read bit at bit 6, and write bit at bit 5. Implicit code regions store the enabled bit at bit 7 and executable bit at bit 6.

Although implicit regions are used for unmodified native binaries, we are still working with the assumption that the software setting up HFI for the binary is a trustable user who will setup the regions properly. However, one check we do insert is for uninitialized regions. Originally, if a region wasn't setup properly where *hfi_set_region_size* wasn't called but *hfi_set_region_permissions* was called, a region would still be configured with $0x0$ for the base and mask. This was an issue as such a region allowed any code/data addresses to pass through. Hence, to prevent this, we set the mask on implicit regions to $0x11111111$ initially which would prevent regions not set up properly to fail HFI checks.

To implement our instruction we have opted to use the free opcode 0001011. We are then able to differentiate between these functions and the two aforementioned functions (*hfi_enter* and *hfi_exit*) by modifying the funct3 value of the instructions such that *hfi_set_region_size* has funct3 $= 010$ and *hfi_set_region_permissions* has funct3 $= 011$.

5.3. Implementation - Explicit Regions

Next, we moved on to implement explicit regions by defining a new struct *HFIExplicitDataRegion* to store metadata about explicit regions. These structs were also stored in a new array in the CPU Architecture State of length based on the global variable indicating the number of regions we wanted to support. While the original HFI paper had the h-prefix *mov* instruction (*hmov*), we had h-prefix load and store instruction. This difference is due to the ISA differences between x86 and RISC-V. Hence, for our implementation, we focused on providing support for a single region and implemented the following instructions: *hlb0*, *hlh0*, *hlw0*, *hlbu0*, *hlhu0*, *hlwu0*, *hld0*, *hsb0*, *hsh0*, *hsw0*, and *hsd0*.

For each of the load and store instructions, we would check whether both the beginning of the access address and end of the address access fall within a given data region (region 0 for the h-prefix and 0-postfix instructions). If HFI is enabled and the load or store fall within a properly configured region, then the load or store will be allowed through and proceed to load or store the given address by passing on the access addresses to the corresponding non-

TABLE 3. ENCODINGS OF HFI INSTRUCTION IMPLEMENTED

	opcode	funct3
hfi_enter	0001011	000
hfi_exit	0001011	001
hfi_set_region_size	0001011	010
hfi_set_region_permissions	0001011	011
hlb0	0101011	000
hlh0	0101011	001
hlw0	0101011	010
hlbu0	0101011	100
hlhu0	0101011	101
hlwu0	0101011	110
hld0	0101011	011
hsb0	1011011	010
hsh0	1011011	011
hsw0	1011011	100
hsd0	1011011	000

HFI load or store instruction. If the check fails, HFI would trap and prevent access to the address.

Explicit data regions are configured using a base and a bound. To set these regions, we use the same *hfi_set_region_base* and *hfi_set_region_permissions* that we used for implicit regions. The value passed for a mask in implicit regions was instead used as the bound value when an explicit region was configured. For the permissions, explicit data regions store the enabled bit at bit 7, read bit at bit 6, write bit at bit 5, and large region bit at bit 4.

We tested the implementation using this single explicit data region. One obvious limitation is how we can scale this to work for multiple data regions, which can be a challenge due to the opcode limitation discussed in Section 4. However, based on the feasibility of our implementation for the one data region and overall structure of the implementation, we believe proposed ideas in Section 6 can allow easy extension to support more explicit data regions.

For these new load instructions, we used the free opcode 0001011. For the new store instructions, we used the free opcode 1011011. You can find the funct3 values we used to differentiate between them in Table 3.

5.4. Evaluation - Testing

To validate our HFI implementation and ensure correct functionality across all features, we developed a comprehensive testing infrastructure. Our testing framework consists of a suite of assembly programs written specifically to test different aspects of HFI functionality on RISC-V. These tests are compiled using the RISC-V GCC toolchain and executed on our modified QEMU binary.

We use a modular testing approach where each assembly program focuses on testing a specific component or feature of HFI. Our test suite covers several key areas of functionality: basic HFI operations that verify the core instructions function correctly and properly toggle the HFI status; region configuration tests that ensure memory regions can be properly defined with appropriate size and permissions; memory access tests that verify operations within configured regions succeed while operations outside them fail appropriately; and explicit region access tests that exercise the h-prefix load/store instructions to validate boundary checking and permission enforcement. We also implemented control tests to establish baseline functionality in non-HFI contexts.

These test programs are written in RISC-V assembly and leverage a custom set of macros defined in our includes directory that encode our HFI instructions using the appropriate opcodes and operands. For example, our HFI enter macro encodes the instruction with opcode 0x0B and function code 0 along with the specified register operands. This approach allows us to write readable assembly that directly tests our custom instructions.

We developed a Makefile-based build system that compiles these assembly programs using the RISC-V toolchain (riscv64-unknown-elf-as and riscv64-unknown-elf-ld). The resulting ELF binaries are then executed on our custom-built QEMU with additional debug flags to capture instruction execution and CPU state information.

The test execution outputs are captured in log files, allowing us to analyze instruction execution, register state changes, and any traps or exceptions that occur during testing. This approach enables us to verify several critical aspects of the implementation: the HFI instructions execute with the expected opcodes, region configurations are properly stored in CPU state, memory accesses are correctly validated against region bounds, and violations trigger appropriate traps when they should occur.

Through this testing infrastructure, we confirmed the correctness of our implementation across various scenarios including successful sandboxing, proper isolation of memory regions, and correct handling of boundary conditions. The modular nature of our test suite allows us to incrementally validate features as they are implemented and regression-test them as the implementation evolves.

5.5. Evaluation - wasm2c

We wanted to additionally add some of the evaluation done by the original HFI paper. We attempted to modify wasm2c by building on the work done already. The wasm2c code converts wasm files to C source and header and we wanted to modify it to use our modified h-prefix load and store instructions. Unfortunately, we were unable to complete this due to some issues with building and running the module. However, this doesn't detract from the implementation of HFI itself. Although it would have been valuable to have used our HFI implementation in wasm2c, our unit tests still comprehensively demonstrate that HFI is feasible and functional in RISC-V. Our implementation is able to achieve the sandboxing functionality we wanted and effectively shows that HFI can extend to other architectures beyond x86. Hence, our work sufficiently shows the feasibility of HFI in RISC-V.

6. Future Work

6.1. Expand Number of Explicit Regions

Previously mentioned, we have currently implemented only a singular explicit region with regards to the number of instructions that we would introduce. In the simplest of implementations, we would just associate a separate instruction for each explicit region, but as we have discussed in Section 4, we only have a finite number of available opcodes.

TABLE 4. INSTRUCTION ENCODING FOR `LW` IN RISC-V

31–20	19–15	funct3	11–7	opcode
imm[11:0]	rs1	010	rd	0000011

TABLE 5. INSTRUCTION ENCODING FOR `SW` IN RISC-V

31–25	24–20	19–15	funct3	11–7	opcode
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011

Therefore, we need to fit the targeted explicit region number within the load and store instructions we create. However, such an observation introduces a larger problem. At closer inspection of the load and store instructions, which are the same as the `lw` and `sw` with varying funct3 values, it can be seen that these instructions are extremely tightly packed. Thus, any attempt at adding region information will remove information from some other component of the instruction. For support of two explicit regions, this will require just 1 bit. Likewise, for supporting four explicit regions as in the original HFI [1] paper, we will need 2 bits. As a result, we propose two potential solutions in pursuit of supporting four explicit regions.

Aligning Accesses: In this method, the 2 bits can be sourced from the immediate with the specification that the added immediate will now have to be a multiple of 4. This works without any problems for the loading and storing of a word (4 bytes) and sizes greater than or equal to 4 bytes, but creates issues when considering half words and byte level modifications. One possible modification is to also align the halfword accesses to multiples of 2 and have two separate instructions. However, for byte level accesses this is not possible. Although this implementation can exclude existing programs that might want to be sandboxed and leverage HFI, in general most code generated from programming languages will be aligned for improved memory access. Moreover, any problems that do occur can be solved via compiler modification to modify the unaligned memory accesses to use a combination of byte, halfword, and word versions of the hfi instructions, although this requires substantially more effort.

Reducing Available Registers: Here, the 2 bits can be sourced from the indexing of the two registers in the relevant instruction. Registers are input into instructions via a 5 bit value as there are 32 registers in total. Therefore, as each of these instructions have two registers in some form (two sources or one source and one destination), we can take a single bit from each of the 5 bits denoting the two registers. This gives us the 2 required bits at the tradeoff of restricting the number of available registers. The range of the available registers can additionally be offset or mapped according to a defined set. The use of this method is potentially viable, but more evaluation should be done to verify how problematic register spilling might become.

In reality, a combination of these two methods will probably yield a better result for saving the number of added instructions for new explicit regions. Additionally, further evaluation could explore encoding strategies and compiler transformations to better support multiple explicit regions without excessive instruction duplication.

6.2. Utilize the RISC-V HFI in SFI

As previously mentioned, a natural next step is to integrate our RISC-V HFI instructions into an existing SFI compiler to validate the correctness of end-to-end sandboxing behavior. This would involve instrumenting memory accesses and control flow in compiler-generated code and replacing them with HFI-enforced operations. Such an integration would demonstrate the correctness of SFI compilers when relying on hardware enforced isolation mechanisms.

Additionally, supporting integration of HFI instructions into existing SFI compilers requires certain specification-focused instructions that we did not implement, such as:

- `hfi_get_version`
- `hfi_get_linear_code_range_count`
- `hfi_get_linear_data_range_count`
- `hfi_get_exit_reason`
- `hfi_get_exit_location`
- *and more...*

These additional instructions can be added with minimal modification to the `funct7` field.

6.3. Cycle Accurate Simulator

A more comprehensive implementation for evaluation of security and performance in addition to ISA semantics can be considered in a cycle accurate simulator, such as gem5, as done in the original HFI paper [1]. Since we used QEMU, an environment where the serialization of *hfi_enter* and *hfi_exit* would not produce observable effects, we chose not to implement serialization for Spectre mitigation. In contrast, a cycle-accurate simulator enables observation of microarchitectural behavior, allowing for both functional correctness validation and evaluation of speculative execution resistance. Moreover, with an accurate cycle simulator, the performance overhead for HFI implementation can be validated when utilized within a SFI system. As an implementation in a cycle accurate simulator is comparatively much slower in code modification and execution as compared to QEMU, as a feasibility metric for correctness, an implementation in QEMU was ideal with respect to our circumstances.

7. Related Work

Our work on HFI for RISC-V builds on prior work that established HFI for x86-64 architectures [1]. HFI provides a simple ISA extension to support secure, flexible, and efficient in-process isolation, addressing limitations of existing software-based techniques, including runtime overheads, scalability limits, Spectre vulnerability, and compatibility issues.

There are also several alternate approaches have been developed over the years for fine-grained isolation involving hardware-assisted techniques. Some systems have proposed extensions to the page table metadata to store a per-sandbox ID checked by hardware, such as Donky [2] or IMIX [4]. Donky is a hardware-software co-design providing strong in-process isolation based on dynamic memory protection domains using hardware-backed protection keys. Its implementation on RISC-V uses the reserved top 10 bits of page-table entries for protection keys, supporting 1024 keys per process. Donky distinguishes itself by managing protection key policies entirely in userspace via a dedicated control and status register (DKRU). Access to the DKRU is protected by a hardware call gate, which leverages the RISC-V N extension. This architecture provides substantially stronger security guarantees than systems like Intel MPK, where the policy register is unprivileged, allowing Donky to shield against arbitrary code execution without needing binary inspection or rewriting. Donky facilitates fast domain switches entirely in userspace, with minimal hardware additions on RISC-V. Donky has been implemented for both x86 (using MPK emulation) and RISC-V architectures, while IMIX has only been implemented on x86.

IMIX (In-Process Memory Isolation Extension) introduces a minimal x86 ISA extension for isolating sensitive data within a process [4]. Unlike many isolation mechanisms tied to a specific defense (e.g., MPK for protection keys or CET for shadow stacks), IMIX provides a mitigation-agnostic hardware primitive for protecting security-critical data. IMIX enables memory pages to be marked as isolated using a new permission flag. These pages can only be accessed with a special instruction, smov, introduced by IMIX. This mechanism allows a wide range of memory-corruption defenses (such as CPI or CFI) to protect their internal metadata from arbitrary access by potentially compromised application code. IMIX was prototyped using Intel's simulation infrastructure, with extensions to LLVM and Linux to support its new instruction semantics. The authors demonstrated that IMIX significantly reduces the attack surface for in-process defenses with low overhead and better flexibility than existing solutions like MPK, which are limited by unprivileged key registers and require complex mitigation workarounds to be secure.

In addition to this, there are also methods that make use of capability-based addressing, a scheme for controlling access to memory, which is seen most prominently in CHERI [3]. While initially developed as a hybrid capability model extending the 64-bit MIPS ISA and demonstrated on FPGA, the CHERI approach is described as not being specific to MIPS and is a prominent capability system relevant to discussions of advanced ISAs, including RISC-V (as referenced in related work comparisons). CHERI provides byte-granularity memory protection and enables hardware enforcement of memory safety and fault isolation. It utilizes a capability coprocessor and tagged memory, where capabilities are stored in dedicated registers and memory locations, protected by hardware tags and unforgeable manipulation instructions. Capabilities are unforgeable and define protection domains based on reachability. CHERI aims for a high level of source-code and binary compatibility relative to prior pure capability machines, supporting incremental deployment. However, adapting and recompiling application code is generally required to utilize its full protection features, and achieving comprehensive object-level memory safety involves extensive modifications throughout the software stack (OS, ABI, compiler). CHERI offers strong fine-grained protection within an address space and supports scalable userspace protection domains.

In summary, while Donky leverages protection keys and a userspace monitor with hardware protection for the policy register, and CHERI introduces capabilities for fine-grained, unforgeable memory references requiring significant software stack changes, IMIX proposes a lightweight ISA extension with minimal hardware cost to securely restrict access to sensitive memory used by diverse in-process defenses. Our work on HFI for RISC-V focuses on a region-based mechanism, minimal hardware modifications, userspace-only operation, and explicit Spectre mitigation for flexible and efficient in-process isolation supporting both adapted and unmodified code.

Acknowledgments

References

[1] Shravan Narayan, Tal Garfinkel, Mohammadkazem Taram, Joey Rudek, Daniel Moghimi, Evan Johnson, Chris Fallin, Anjo Vahldiek-Oberwagner, Michael LeMay, Ravi Sahita, Dean Tullsen, and Deian Stefan. 2023. Going beyond the Limits of SFI: Flexible and Secure Hardware-Assisted In-Process Isolation with HFI. In Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3 (ASPLOS 2023). Association for Computing Machinery, New York, NY, USA, 266-281. https://doi.org/10.1145/3582016.3582023

[2] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys-Efficient In-Process Isolation for RISC-V and x86. In Proceedings of the USENIX Security Symposium (USENIX Security).

[3] Jonathan Woodruff, Robert N.M. Watson, David Chisnall, Simon W. Moore, Jonathan Anderson, Brooks Davis, Ben Laurie, Peter G. Neumann, Robert Norton, and Michael Roe. 2014. The CHERI Capability Model: Revisiting RISC in an Age of Risk. SIGARCH Comput. Archit. News 42, 3 (2014). https: //doi.org/10.1145/2678373.2665740

[4] Tommaso Frassetto, Patrick Jauernig, Christopher Liebchen, and Ahmad-Reza Sadeghi. 2018. IMIX:In-Process Memory Isolation EXtension. In Proceedings of the USENIX Security Symposium (USENIX Security).

[5] About qemu (no date) About QEMU - QEMU documentation. Available at: https://www.qemu.org/docs/master/about/index.html (Accessed: 26 March 2025).

[6] Performance comparison between models and other simulators We evaluated the performance. Retrieved from https://www.researchgate.net/figure/Performance-comparison-between-models-and-other-simulators-We-evaluated-the-performance-fig2-341640101

[7] Anon. Adding a custom instruction in the RVI subset. Retrieved March 26, 2025 from https://pcotret.gitlab.io/riscv-custom/adding-custom.htmlcustom-opcodes

[8] "V Ratified Specifications." RISC, riscv.org/specifications/ratified/. Accessed 30 Apr. 2025.