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
University of Texas at Austin
stanley.p.joseph@gmail.com

***Abstract*—**

1. Introduction

1.1. 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. 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. 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.

1.2. Background

Although hardware-based isolation solutions can be integrated at multiple levels, our approach 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. Concretely, existing in-process isolation techniques—such as WebAssembly-based solutions—often rely on software-based guard pages or software fault isolation (SFI). These approaches can impose substantial memory footprints and performance overheads, particularly in tight loops or heavily sandboxed server-side environments. Moreover, software guard pages require complex interactions with the operating system's memory manager (e.g., mprotect calls) that slow down sandbox creation or resizing.

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. 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.

1.3. Security/Performance of Software-only Isolation

Software-only isolation has proven susceptible to exploits that take advantage of subtle boundary checks or speculative execution vulnerabilities. Past disclosures—ranging from the Heartbleed bug (CVE-2014-0160), exposing sensitive in-memory data, to speculative execution attacks like Spectre—highlight that user-level components can become an attack vector if not properly contained. 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.

1.4. Approach

Our work brings HFI concepts to the RISC-V ecosystem by introducing new "region"-based memory restrictions, a dedicated sandbox mode, and fast transition instructions—all while fitting into RISC-V's existing privileged architecture. Specifically, we rely on the following techniques:

1) In-Process Isolation via Regions. Inspired by previous x86-64 prototypes, 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, we avoid ring transitions for memory setup. 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. We adopt 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.

By systematically combining these hardware instructions, metadata registers, and user-space management strategies, 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.

2. Progress

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