# Going beyond the Limits of SFI: Flexible and Secure Hardware-Assisted In-Process Isolation with HFI

Shravan Narayan
UC San Diego, USA
UT Austin, USA

Tal Garfinkel
UC San Diego, USA

Mohammadkazem Taram
Purdue University, USA

Joey Rudek
UC San Diego, USA

Daniel Moghimi
UC San Diego, USA

Evan Johnson
UC San Diego, USA

Chris Fallin
Fastly, USA

Anjo Vahldiek-Oberwagner
Intel Labs, Germany

Michael LeMay
Intel Labs, USA

Ravi Sahita
Rivos, USA

Dean Tullsen
UC San Diego, USA

Deian Stefan
UC San Diego, USA

## ABSTRACT

We introduce Hardware-assisted Fault Isolation (HFI), a simple extension to existing processors to support secure, flexible, and efficient in-process isolation. HFI addresses the limitations of existing software-based isolation (SFI) systems including: runtime overheads, limited scalability, vulnerability to Spectre attacks, and limited compatibility with existing code. HFI can seamlessly integrate with current SFI systems (e.g., WebAssembly), or directly sandbox unmodified native binaries. To ease adoption, HFI relies only on incremental changes to the data and control path of existing high-performance processors. We evaluate HFI for x86-64 using the gem5 simulator and compiler-based emulation on a mix of real and synthetic workloads.

## CCS CONCEPTS

• **Security and privacy** → **Hardware security implementation**; **Operating systems security**; *Browser security*.

## KEYWORDS

SFI, Wasm, sandboxing, hardware-based isolation

**ACM Reference Format:**
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 '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3582016.3582023

*ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada*

ACM ISBN 978-1-4503-9918-0/23/03.
https://doi.org/10.1145/3582016.3582023

## 1 INTRODUCTION

WebAssembly (Wasm) [28] has made in-process isolation ubiquitous. In the browser, it powers applications used by billions of people daily [49, 64, 79]. Beyond the browser, it enables isolation in places that existing hardware-based protection can't — from hyper-consolidated FaaS platforms [32, 76] and high-performance data planes [59], to data streaming platforms [61].

Wasm makes these novel use cases possible by enforcing isolation in software — using software-based isolation (SFI) — which avoids the high overheads imposed by existing hardware-based isolation primitives [54, 71] (e.g., processes, containers, and VMs). This approach to isolation enables several unique properties.

To start, Wasm context switches are very fast — in the low 10s of cycles [38], roughly the same as a function call — and orders of magnitude cheaper than a hardware context switch [30], let alone IPC. These fast context-switches let Wasm provide extensibility in high-performance data planes [59], data streaming platforms [61], and SaaS applications [56]; they also enable fine-grain isolation of vulnerable libraries in latency sensitive browser renderers [10, 52].

Wasm context creation is also very fast — production FaaS systems can spin up a new Wasm instance in 30 $\mu$s [20], instead of the tens to hundreds of milliseconds it takes to spin up a container or VM [20, 48, 69]. Along with low context-switch overheads, this has enabled a new class of high-concurrency, low-latency edge computing platforms from Fastly [32], Cloudflare [76], Akamai [2], etc. that would not have been possible with containers or VMs.

Unfortunately, the power of software-based isolation also comes with limitations: *Performance* — even the fastest Wasm implementations can easily impose a 40% overhead on code execution [36, 88] limiting Wasm's ability to support more demanding workloads; *Scaling* — Wasm relies on an ad-hoc system of guard regions for memory isolation (§2) which consumes huge amounts of virtual memory and limits efficiency in high-scale settings like FaaS platforms; *Backwards compatibility* — Wasm cannot run unmodified binaries (e.g., system libraries), code that directly accesses hardware (e.g., SIMD intrinsics, assembly language), or dynamically generated code (e.g., from just-in-time compilers); *Spectre safety* — processors can speculate past security checks in Wasm making it vulnerable to Spectre attacks[1]. These limitations are the result of trying to bridge, in software, the gap between past models of hardware protection and the current needs of software systems.

To overcome them, we developed *hardware-assisted fault isolation* (HFI) — a simple ISA extension that brings support for in-process isolation to modern processors.

HFI takes a two track approach to support in-process isolation (aka sandboxing). First, it provides hardware assistance to eliminate the limitations Wasm inherits from SFI — essentially replacing SFI with efficient hardware primitives. Second, HFI provides in-process isolation that is backwards compatible, allowing it to sandbox unmodified native binaries and dynamically generated code.

HFI offers primitives that systematically eliminate typical hardware (and software) overheads by design: it imposes near-zero overhead on sandbox setup, tear-down, and resizing; it can support an arbitrary number of concurrent sandboxes; it offers context switch overheads on the same order as a function call; it can share memory between sandboxes at near-zero cost; it provides flexible low-cost mitigations for Spectre, and near-zero cost system call interposition (for native binaries).

We made a few key choices that enable this. First, HFI does everything in userspace; thus, there are no overheads from ring transitions or system calls when changing memory restrictions, or entering and leaving a sandbox. Second, HFI does not rely on the MMU for in-process isolation — instead, sandboxing is enforced via a new, orthogonal mechanism called *regions* (§3.2); regions enable coarse-grain isolation (e.g., heaps) and fine-grain sharing (e.g., objects) in the processes' address space. Third, HFI only keeps on-chip state for the currently executing sandbox; thus, it can scale to an arbitrary number of concurrent sandboxes — in contrast, many other systems hit a hard limit as they keep on-chip state for all active sandboxes [9, 25, 27, 60, 66, 75].

Beyond this, there are many seemingly small overheads in common operations whose cumulative impact can be large; we sought to minimize these costs. Our design eliminates unnecessary context switches for Wasm sandboxes (§3.1), and lets software choose the most efficient mechanism for implementing context switches (§3.3). Spectre mitigations can add overhead by serializing instructions, so we devised a flexible mechanism to minimize the need to serialize (§3.4). Finally, system call interposition can be complex and expensive when dealing with many concurrent sandboxes; thus, we developed a simple low-cost mechanism to enable this (§6.4.1).

HFI reduces the barriers to in-process isolation, allowing it to be deployed more pervasively and at scale. It achieves this with minimal additional hardware, and minor changes to the control and data paths of existing processors.

To evaluate our design, we implemented HFI twice for x86-64 (§5.2). First, we developed a gem5 simulation to enable detailed performance analysis. Next, we built a compiler based emulator that approximates HFI overheads in target workloads; this allows us to run larger and more complex workloads than would be possible in gem5. To ensure accuracy, we validated the precision of our emulator using our gem5 simulation.

We integrated HFI emulation into the Wasm2c ahead-of-time compiler and the Wasmtime just-in-time compiler (§5.1), and evaluated its performance on the SPEC CPU 2006 benchmarks (§6.1), sandboxed font and image rendering in Firefox (§6.2), and a simplified FaaS setting (§6.3). We also applied it to sandboxing native code (OpenSSL) in the NGINX webserver (§6.4.2)[2].

Our results show that HFI-assisted Wasm achieves strictly better performance than stock Wasm, and offers noticeable speedups on real workloads, such as a 14%–37% improvement for image rendering in Firefox (§6.2). Our native workloads run at bare-metal speed — modulo the optional (§3.4) added cost of serializing sandbox entries and exits for Spectre protection.

In our FaaS workloads, adding Spectre protection with HFI leads to a 0%–2% increase in tail latency as compared to unsafe native execution. In contrast — adding Spectre protection using Swivel [53], the fastest known software-based mitigation, leads to a 9%–42% increase in tail latency (Table 1).

Most of Wasm's limitations stem from its reliance on software techniques for memory isolation; we explore these in the next section. We then present HFI, and how it lets us go beyond these limitations (§3). We explore how HFI accomplishes this securely, and with minimal hardware at a microarchitectural level (§4). Finally, we evaluate HFI (§6), survey related work (§7), and offer conclusions.

## 2 LIMITATIONS OF SFI

The limitations of modern page-based protection architectures for fine-grain isolation are well known [54]. They include expensive context switches due to protection ring transitions, heavy weight saves and restores [30], increased TLB flushes and contention as concurrency scales etc.

SFI [78] — and by extension, Wasm — avoids these costs by instead relying on compiler-added instrumentation to enforce isolation. However, simply adding conditional bounds checks to each memory load/store and instruction fetch can easily slow down code by a factor of 2× [78, 88]. Past SFI compilers instead relied on the (somewhat faster) technique of applying a bit mask to the addresses used by loads/stores/etc., prior to their use, thus forcing data access and control flow into the appropriate portion of address space. However, as a side-effect, this converts out-of-bounds memory accesses into (seemingly random) memory corruption.

Consequently, Wasm and other modern SFI systems [68, 86] rely on a faster and safer technique—using the MMU to enforce memory bounds implicitly — as a poor man's version of segmentation.

To accomplish this, a Wasm runtime sets aside an 8 GiB memory region per sandbox, 4 GiB for the sandbox address space, followed by a 4 GiB guard region (unmapped address space). It reserves this space by `mmap()`'ing the entire 8 GiB without permissions.

Additionally, the Wasm compiler restricts the format of memory operations to *load(address, offset)* where *address* is a register with a 32-bit value, and *offset* is a 32-bit immediate (constant). On a memory access, the Wasm compiler adds the *address* and *offset*, resulting in a maximum value of $2^{33} - 2$. It then adds this result to the base-address of the address space (the "heap base"), and performs the load. Since 8 GiB ($2^{33}$) has been reserved after the

[1] Attempts to mitigate Spectre today requires prohibitively expensive compiler techniques [53], or complex workarounds that move the Wasm VM into a process [67] — and, unsurprisingly, neither approach is used in mainstream Wasm engines.

[2] All artifacts of our evaluation are available at https://github.com/PLSysSec/hfi-root

heap base, the load either lands in the sandbox address space and continues, or in the guard region and traps.

Finally, a Wasm program can only access the portion of address space it has explicitly requested from the runtime (e.g. using memory_grow() — similar to `sbrk()`). To enforce this constraint, the runtime grants access to the accessible portions of memory by setting page permission using `mprotect()`. Thus, any memory access beyond the end of the heap will trap.

To isolate control flow, Wasm does not rely on any of the above tricks, and instead relies on software control flow integrity [28].

Despite this clever design, Wasm still has many limitations, some fundamental to SFI, and others specific to Wasm's design:

**32-bit address spaces.** The approach above only supports 32-bit address spaces on 64-bit architectures — to support larger Wasm sandboxes [3], or smaller processors, requires old-school SFI masking or conditionals [78]; masking is out (Wasm requires precise trap semantics), and again conditionals are easily a 2× slowdown [71, 88].

**Performance overheads.** Wasm can easily impose performance overheads of 40% — sometimes less, and sometimes a lot more [23, 36]. Some costs are fundamental to SFI, such as restrictions on the formats of memory instructions and added register pressure [71]. Some are specific to Wasm, such as the cost of software CFI, and limited access to SIMD instructions.

Another source of overhead that shows up at scale is the cost of creating and destroying sandboxes. In particular, unmapping memory incurs a TLB shootdown. In FaaS platforms, where sandboxes are constantly being created and destroyed on every incoming network request, this can significantly harm performance.

**Spectre.** Wasm cannot protect itself against Spectre attacks without performance penalties — to wit — software-based mitigations add an additional 62% to 100% of overhead [53] (mitigations relying on CPU re-design to eliminate Spectre fare better [45, 82, 85, 87], but entail overheads and implementation complexity that makes them unlikely to see deployment).

**Virtual memory consumption.** Wasm's guard pages have a large virtual memory footprint that results in several challenges.

8 GiB *is a lower bound* — previously we noted that every Wasm instance consumes 8 GiB of virtual address space — even if it uses just a few megabytes. However, this is actually a lower bound. Popular Wasm runtimes support multiple memories per-instance [4] (e.g., for sharing data between instances) — and these can increase an instances resource footprint by another 8 GiB per-memory. The next generation of Wasm standards [26] promises to further increase this footprint by supporting Wasm applications composed of multiple components, where each library in the application would be a separate component, each with its own memory(s).

*Virtual address space is finite* — typical Intel x86-64 CPUs provide $2^{47}$ (128 TiB) worth of user level virtual address space[3]—which seems like a lot. However, that can be used up surprisingly quickly. For example, if we assume the best case — that each Wasm instance only consumes 8 GiB — then we can run at most 16K ($2^{14}$) Wasm instances concurrently. In FaaS platforms, that spin up a new sandbox in 10s of $\mu$-seconds [20] for every incoming network request, 16K instances is not a large number. Worse, FaaS functions may not finish immediately — for example, they might make HTTP requests (and block). Thus, an address space can fill up quite quickly.

*Operations systems are slow* — FaaS systems could of course handle scaling limits by spinning up multiple processes and load balancing requests between them — relying on the OS to context switch threads between processes, and context switch processes once these exceed the number of physical cores, and so on. However, the main reason FaaS providers use Wasm is to avoid these overheads in the first place. FaaS providers would rather schedule more instances in fewer processes — ideally one. If used efficiently, 128 TiB really does support a lot of Wasm instances; not only is this more efficient, it makes systems easier to understand, which in turn makes them easier to deploy, debug, and optimize.

Finally, applications that use FaaS platforms don't always consist of just one function, they can be multiple functions that want to communicate (function chaining). In a single address space, this communication is as fast as a function call, however, this is easily 1000× to 10000× slower across process boundaries (IPC) [30, 38].

In the next section, we explore HFI, our hardware extension that addresses these limitations.

## 3 THE HFI DESIGN AND INTERFACE

Hardware-assisted fault isolation (HFI) is our extension for modern processors that supports flexible in process isolation, and offers the following properties:

(1) **Security.** HFI provides all the capabilities needed for secure sandboxing of Wasm and native binaries, including data and control flow isolation (§3.2), complete mediation of the OS interface including system calls (§3.3), and Spectre mitigation (§3.4).

(2) **Efficiency.** HFI imposes minimal overhead for critical operations. Memory isolation with HFI imposes no overhead — all memory bounds and permission checks execute in parallel with TLB lookups (§4.2); Context switches are software managed; thus Wasm can exploit *zero-cost* techniques [38] to optimize them (§3.3.1); System call interposition is near zero cost — HFI converts system calls into jumps (§4.4); HFI's overhead for creating and destroying sandboxes, and sharing and resizing sandbox memory is near zero — only requiring a few HFI instructions to update region registers (§4.4)[4]. Spectre protections are flexible and configurable, allowing developers to avoid unnecessary serialization (§3.4).

(3) **Scalability.** HFI imposes no limit on the number of concurrent sandboxes a program can run — it achieves this by keeping the amount of on-chip state constant, regardless of the total number of sandboxes (§4).

(4) **Compatibility.** HFI supports the unique requirements of Wasm and native binaries. For Wasm, HFI offers precise memory fault semantics (i.e., out-of-bound memory access traps), granular heap growth (64K increments), code and data separation, and direct access to system calls for Wasm sandbox runtimes (via. sandbox types (§3.3)). For unmodified native binaries, HFI eschews any changes to compilers, standard libraries, or binary formats (i.e., ABI changes), and supports simple and efficient system call interposition.

[3] Intel supports 52/57-bit address spaces in certain high-end server CPUs.

[4] To be clear, HFI only does isolation, not resource management — thus, additional sandbox creation overheads, like memory allocation, are up to the developer.

(5) **Adoptability.** For easy adoption, HFI minimizes changes to existing operating systems and processors. OS kernels only need to add a small amount of per-process storage to save HFI's registers during a process's context-switch. To support the OS, HFI extends the processor instructions that save and restore a process's registers (`xsave` and `xrstor` on x86) to include HFI's registers (§3.3.3). Existing processors require only a small amount of additional hardware, and minimal changes to data and control paths to support HFI (§4).

We start with a high-level overview of HFI in §3.1. We then take a deep dive into regions — HFI's mechanism for controlling access to memory in §3.2. In §3.3, we explore HFI's other features in the context of implementing a sandboxing system and see how HFI mitigates Spectre in §3.4. For reference, the complete HFI interface is listed in the appendix.

## 3.1 HFI Overview

HFI allows developers to build sandboxing runtimes that can efficiently create multiple in-process sandboxes each with its own view of process memory.

HFI's interface is accessible entirely in user space — there is no kernel component or ring transitions. Thus, a runtime can rapidly instantiate sandboxes, share memory, etc. HFI can support a wide range of uses cases, from sandboxing untrusted libraries in a large applications [52], to supporting Wasm sandboxes in a FaaS platform.

We refer to a runtime managing sandboxes alternately as the runtime or trusted runtime, since in our HFI threat model, we assume this code is trusted, and the sandboxed code is untrusted — although things get slightly more nuanced with hybrid sandboxes (see below). HFI builds on a few central concepts:

**HFI mode.** Each CPU core has its own HFI state, stored in registers. If HFI is enabled, code running on that core is "sandboxed", i.e, execution is restricted according to: (a) a set of region registers (that grant access to memory), (b) a register with the sandbox exit handler (where system calls and sandbox exits are redirected to), and (c) a register with sandbox option flags (e.g., the sandbox type). With a few exceptions, HFI is enabled when the trusted runtime executes an `hfi_enter` instruction, and disabled when sandboxed code executes an `hfi_exit` instruction — which transfers control back to the trusted runtime. The runtime is responsible for saving and restoring context appropriately, and can use HFI to multiplex many sandboxes across cores, scheduling them as it sees fit.

**Interposition.** HFI supports interposition on all paths out of the sandbox including system calls (and by extension, signals), and sandbox exits (`hfi_exit`). Thus, the runtime can completely mediate [63] the interaction of sandboxed code with the operating system and enclosing process. Supporting interposition directly in HFI, rather than relying on existing OS mechanisms [65, 75] offers excellent flexibility, reduces complexity (which benefits security [14]), improves performance, and eases deployment.

**Sandbox types.** HFI supports two sandbox types: *native* to support standard in-process isolation, and *hybrid* to support Wasm and other SFI systems. The key difference between these two is their trust model. In native sandboxes, HFI assumes sandboxed code is untrusted; in hybrid sandboxes, HFI assumes sandboxed code is trusted — or more specifically, that it was built with a trusted compiler (or checked with a trusted verifier [37, 50, 86]). Because HFI knows hybrid sandboxes are trusted, it allows privileged operations such as system calls and sensitive register updates. This allows hybrid sandboxes to avoid sandbox exits (with the resulting cost of context switches) entirely, modulo whatever the runtime is doing to multiplex HFI. We explore this further in §3.3.

**Regions.** In HFI mode, all memory access is controlled using *regions*. Conceptually, these can be thought of as an address range described by a base (start address for the range), a bound (the size of the range), and a set of permissions (read, write, and execute). In general, every sandbox will have a set of *data regions* (e.g., for its heap, stack, and shared data) and *code regions*. Regions are an attractive representation as they require minimal state (it is easy to save/restore for fast context switches) and can be enforced with simple hardware. As regions are a central feature of HFI, we will explore them first.

## 3.2 Isolating Data & Control Flow with Regions

By default, a sandbox has no access to memory — it cannot read data or run code. A runtime grants access to memory using *regions*, by configuring the *region registers* prior to sandbox entry. HFI offers two types of regions, implicit and explicit, each specialized to different tasks:

**Implicit regions.** Implicit regions apply checks to every memory access, and grant access on a first-match basis. For example, if sandboxed code executes an instruction — *load address X into register Y* — HFI will check if *any* region register has a range that includes $X$ in parallel, then apply the permissions from the first matching region. If the first match has read permission, the operation will proceed, and if it does not, HFI will trap.

Implicit regions are essential for isolating unmodified native code, and similarly, for isolating control flow, situations where explicit regions (described next) would be impossible to use.

Implicit regions perform efficient bounds checks based on *prefix matching* (§4). Concretely, each region specifies a *base_prefix* (the region's base address) and an *lsb_mask*. To check if an address is in bounds, HFI uses the *lsb_mask* to remove the least significant bits of the address, and compares the remaining prefix to *base_prefix*. With this approach, implicit regions must be power of two sized and aligned — thus, they trade granularity for efficient checking.

HFI discriminates implicit regions into code and data regions, to keep the control and data pipelines simpler and more efficient. Thus, data region checks apply only to reads and writes, while code regions check apply only to instruction fetches.

HFI provides six implicit regions per-sandbox, four data regions (e.g., for the heap), and two code regions (e.g., for the application and shared library code)[5]. Implicit regions checks are not applied to operations on explicit regions, which we discuss next.

**Explicit regions.** An explicit region acts as a handle to a memory range, and follows the normal (base, bound) style of addressing. Thus, addressing is always relative to the base of a specified region. For example, if a sandbox executes an instruction: *read address X in region1 into register Y*. If $X < region1[bound]$ is true, and reads are permitted. HFI will store the contents of $region1[base\_address] + X$ into $Y$, otherwise it will trap.

[5] The region count was based on experience sandboxing code in production settings.

HFI provides four explicit regions, and two different region sizes (*large/small*), with different granularities. Large regions can address up to 256 TiB ($2^{48}$) and are sized and aligned to multiples of 64K ($2^{16}$). Small regions, in contrast, can only address up to 4 GiB ($2^{32}$), but are byte granular in size and alignment. Small regions have one additional restriction: they cannot span addresses which are multiples of 4 GiB.

We note that, while allowing regions which support arbitrary address ranges at any grain is conceptually simpler than specialized large and small regions, our restrictions allow bounds checking with very simple hardware. HFI's large and small regions constraints can be checked with a single 32-bit comparator, rather than the multiple 64-bit comparators needed to check arbitrary region bounds (§4.2).

Explicit regions' added granularity is critical for supporting Wasm heaps[6], which grow in 64K increments [28] — while byte granularity is critical for efficiently sharing individual memory objects and sandboxing legacy code, as existing buffers can be shared in-place changing code or allocators.

Explicit regions are accessed using the `hmov` instruction. There are four `hmov` instructions `hmov{0-3}`, one to access each of the explicit regions. For example, `hmov0` is used to access the region specified by the first region register. To ease adoption in existing compilers, `hmov` offers the same semantics as the normal x86 `mov` instruction — with the following caveat.

Unlike the normal `mov`, `hmov` ensures that only positive offsets of explicit regions are accessed — a guarantee necessary for a simple implementation in the hardware (§4). To elaborate, the normal x86 `mov` takes multiple operands which are added to generate the *effective address* for a memory operation. The `hmov` instruction modifies this in the following ways: (1) the first operand is always ignored and replaced with the specified region's base address, (2) `hmov` traps if a negative value is used for the remaining operands, and (3) `hmov` traps if the effective address computation overflows.

While some aspects of these restrictions may seem onerous at first glance, they only rule out patterns that compilers do not rely on in-practice, or can easily work around; for instance, overflows in effective address computations are undefined behavior in C/C++.

## 3.3 Sandboxing with HFI

Next, we explore how HFI's features are used to instantiate and run sandboxes. As a motivating example, we will assume we are a function-as-a-service (FaaS) provider building a trusted runtime to sandbox client applications. In our example, our FaaS can support both Wasm applications and native applications.

*3.3.1 Entering a Sandbox.* Let's assume we've gotten a network request, and our runtime is ready to start a sandbox — application code is in memory, heap space is allocated, inputs (e.g., the headers and body of an HTTP request to our FaaS service) are in a buffer in memory. Our runtime can now take the following steps:

**Setting up regions.** To start, our runtime sets up access to the code, heap, and input memory so our application has everything it needs once the sandbox starts. The runtime does this by using the `hfi_set_region` instruction which stores these initial region mappings into the specified region's registers. Notably, if no code regions are mapped, HFI will immediately trap after `hfi_enter` is called, as the processor will not be able to fetch instructions.

**Selecting a sandbox type.** Our runtime selects a sandbox type (hybrid/native), which it passes as a flag to `hfi_enter`. If the runtime is running untrusted code, it chooses the native sandbox, causing HFI to *lock* all region registers from when `hfi_enter` is called until the sandbox exits, and redirect all system calls to our runtime's exit handler (see below).

If it is running a Wasm application, it chooses the hybrid sandbox, which leaves the region registers unlocked. So, when the Wasm runtime inside the sandbox starts up, it sets up an explicit region that points to the Wasm heap using `hfi_set_region`. As the Wasm code runs, the runtime inside the sandbox can make any system calls it needs to directly. It can also resize the heap, or multiplex HFI's (finite) registers among a larger number of multi-memories [4]. Consequently, scheduling reasons aside, there should be no need to hand control back to the (external) trusted runtime until the application exits.

**Saving context.** Our runtime must protect its own execution context such as its stack and contents of CPU registers, before it switches to sandbox code. Unlike previous systems [15], HFI leaves this mechanism entirely up to software — this flexibility is important for efficiency. For example, if our runtime is running untrusted native code — it will have to use springboards and trampolines [86] — lightweight assembly routines that (1) clear registers and switch to a separate stack prior to executing the sandboxed code and (2) restore these registers after the sandboxed is executed. However, if it is running Wasm code, it could opt to use zero-cost transitions [38] that rely on the compiler to ensure that the sandbox code cannot misuse the stack or scratch registers.

**Setting up an exit handler.** If our runtime is using a native sandbox, it will install an exit handler to take control when `hfi_exit` is called, or when system calls are made in a native sandbox. To install an exit handler, our runtime specifies a function pointer that must be invoked on sandbox exit as parameter to `hfi_enter`. When the exit handler is called after the sandbox exits, it will transfer control to our runtime, which will check a model specific register (MSR) to identify the cause of the exit, and respond appropriately. If HFI is running a Wasm (hybrid) sandbox, our runtime typically will not install an exit handler (though it optionally can), as it is not interposing on system calls and `hfi_exit` instructions; this is safe since the code in the sandbox is trusted.

Having taken all these steps, our runtime is ready to start the sandbox. Once it calls `hfi_enter`, HFI mode is enabled, and the next instruction that runs will be inside a sandbox.

*3.3.2 Leaving a Sandbox.* After the execution of sandboxed code, HFI is disabled, and control is returned to our trusted runtime through a few different paths:

**`hfi_exit` and system calls.** If sandboxed code calls `hfi_exit`, HFI will record the reason for the exit in an MSR, atomically disable HFI mode, and transfer control to our trusted runtime. For native sandboxes, our runtime's exit handler is invoked on exit; for hybrid sandboxes, our runtime will not use an exit handler and instead

[6] Regions make resizing heaps orders of magnitude faster than current Wasm implementations, as regions can be resized with just a register update. In contrast, current Wasm systems use `mprotect()` to limit access to only what the sandbox has requested — thus a system call is always required to resize memory.

allows hfi_exit to fall through to other trusted code placed directly after the exit (allowing an exit handler code to be inlined when using a trusted compiler). The native sandbox's interposition of system calls is nearly identical to an hfi_exit with a handler; system calls are simply converted into a jump to the exit handler by HFI, resulting in very efficient interposition. Again, the cause of the exit, including which system call and type of call (e.g., int 0x80 vs. sysenter) is recorded in an MSR that can be read by the exit handler.

**Access violations and hardware faults.** If sandboxed code causes a hardware trap (e.g., when dereferencing a null pointer), or an HFI bounds check violation (accessing memory outside of the HFI regions) — HFI disables the sandbox mode, records the cause of the fault in a model specific register (MSR), and generates a hardware trap — which the OS delivers as an OS signal to our trusted runtime. For example, upon encountering an HFI bounds check violation, HFI disables the sandbox and generates a fault that is delivered as a SIGSEGV signal to a signal handler that our runtime registered. The signal handler can examine the MSR to disambiguate the cause of the SIGSEGV, and take the next appropriate action.

#### 3.3.3 *OS Support.*

Multiple processes can use HFI concurrently. To enable this, the OS must save the contents of HFI registers (along with the general-purpose registers) when switching between processes. To support this, HFI adds a flag (save-hfi-regs) to the x86 xsave and xrstor instructions, that are used to save and restore process context. Enabling this flag in the kernel is a simple and minimal change. Since this flag modifies the HFI registers, allowing code in a native sandbox to execute xrstor with this flag could break sandboxing; thus HFI will traps if this occurs.

## 3.4 Mitigating Spectre

By construction, HFI prevents whole classes of attacks that could be used to speculatively read data outside the sandbox, since HFI's region checks are applied uniformly to all memory accesses (speculative and non-speculative) once HFI is enabled. However, an attacker could still try to trick the trusted runtime into speculatively running malicious code without HFI enabled — or to speculatively enable HFI in an inconsistent state; we address these risks next.

**Serializing hfi_enter and hfi_exit.** A simple way to mitigate the previously mentioned attacks is to fully serialize hfi_enter and hfi_exit. Serializing hfi_enter ensures that when we enter a sandbox, our configuration is in a consistent state — for example, that some region register does not (speculatively) contain unsafe parameters due to speculative execution, data value prediction, etc. Serializing hfi_exit ensures that malicious code cannot speculatively disable HFI, and then speculatively execute a code path that would never happen under non-speculative execution.

To serialize hfi_enter and hfi_exit, a runtime can set the is-serialized flag on sandbox entry. We expect this to add ≈ 30 − 60 cycles on x86-64 based on the cost of similar serializing instructions [30, 75]; this cost is amortized in many workloads, as we see in §6. However, for applications that don't want to pay this cost, we offer an optional extension to HFI called switch-on-exit, that avoids most of this overhead, but still offers Spectre protection.

**The switch-on-exit extension.** Often there is no need to Spectre isolate a collection of sandboxes — or the same sandbox — across invocations. For example, multiple invocations of a sandbox working on the same data (e.g., for image rendering as discussed in §6.2) are a common occurrence. Other times, a developer knows there are no secrets among sandboxes that need Spectre protection, e.g., among components [26] in an application or FaaS invocations of the same tenant. In these cases, serializing every entry and exit is needlessly expensive. To avoid this overhead — HFI optionally provides the switch-on-exit extension that enables runtimes to restrict speculative control flow to a common set of sandboxes — and ensure that any entry and exit from this set is serialized.

To use switch-on-exit, a trusted runtime must start by running itself in a hybrid sandbox with the *is-serialized* flag set; thus, both its entry and exit will be serialized. This ensures that any control flow into the sandbox cannot speculate beyond its (serialized) hfi_exit. Once this foundation is set, the trusted runtime can run other sandboxes by invoking hfi_enter (unserialized) with the *switch-on-exit* flag set — doing so will save the trusted runtime's HFI registers, and atomically switch to the registers of the new sandbox. Sandboxes started this way cannot disable HFI when they exit (e.g., with hfi_exit); instead, HFI will atomically switch back to the trusted sandbox by restoring its registers to the state prior to hfi_enter. Thus, we can run multiple sandboxes without serialization. Ultimately, serialization will take place when we leave the trusted sandbox; thus, an attacker can never speculatively run code outside this collection of sandboxes with HFI disabled.

**Securing the runtime.** Sandbox runtimes execute in close proximity to untrusted code. In the case of Wasm, the runtime even executes within the same sandbox, making it acutely vulnerable to Spectre attacks. Consequently, runtimes need a way to prevent themselves from being tricked into leaking privileged data [53].

Implicit regions are the tool that runtimes in hybrid sandboxes can use to solve this problem. Implicit regions provide a "safety net" for runtime code, such that even if runtimes are under the speculative influence of an adversary, they can secure themselves by constraining their memory accesses to a *safe* portion of the address space. Runtimes of native sandboxes can also leverage this approach by executing within a dedicated hybrid sandbox.

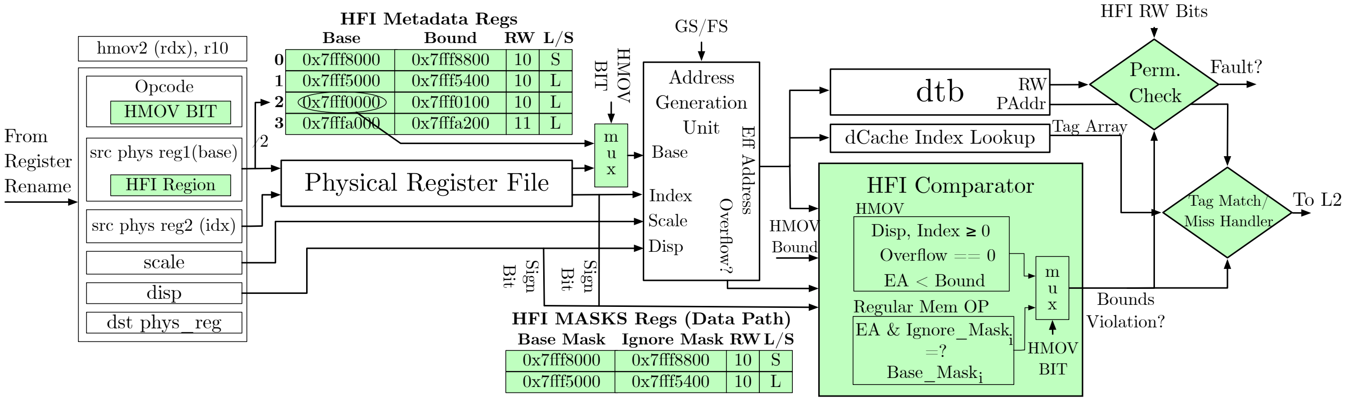
A final important caveat is that using HFI still requires a holistic approach to Spectre security. Running code in a sandbox does not change the influence that malicious code exerts on prediction hardware state, which could influence other software on the system. As a corollary, a runtime that is sandboxing with HFI must ensure that it does not allow a speculative bypass of hfi_enter altogether, followed by speculative execution of untrusted code

# 4 $\mu$ARCHITECTURE DESIGN

HFI's $\mu$-architectural design is guided by four overarching goals:

(1) **Fast.** Minimizing overhead is a central goal of HFI.

(2) **Secure.** HFI must be robust to Spectre-style attacks, and free from Meltdown-style flaws that could compromise existing software. To avoid this, HFI must not update microarchitectural state (such as the dtb, branch predictor, and data/instructions caches) based on data that is secret (i.e., data outside the sandboxed region).



HFI Metadata Regs

|   | Base | Bound | RW | L/S |
|---|---|---|---|---|
| 0 | 0x7fff8000 | 0x7fff8800 | 10 | S |
| 1 | 0x7fff5000 | 0x7fff5400 | 10 | L |
| 2 | 0x7fff0000 | 0x7fff0100 | 10 | L |
| 3 | 0x7fffa000 | 0x7fffa200 | 11 | L |

HFI MASKS Regs (Data Path)

| Base Mask | Ignore Mask | RW | L/S |
|---|---|---|---|
| 0x7fff8000 | 0x7fff8800 | 10 | S |
| 0x7fff5000 | 0x7fff5400 | 10 | L |

**Figure 1: HFI Impact on the x86 data pipeline. We see the x86 data pipeline with added HFI components in Green. HFI adds no additional overhead to the data path—no new pipeline stages are added — and all new operations take place in parallel with the dtb (dTLB) lookup or instruction decode stages.**

(3) **Scalable.** HFI must not constrain the number of concurrent sandboxes. Thus, we eschew designs that store per-sandbox state (such as region details) for multiple sandboxes in on-chip caches such as the TLB. Hardware extensions that do this impose scaling limits either by restricting the total number of concurrent sandboxes [66, 75], or requiring expensive state spills on overflow [55], resulting in performance that scales down as concurrency increases.

(4) **Minimal.** HFI should not significantly change the processor design or add expensive components. This helps generality as minimal designs are easier to implement in small power-conservative chips, and also benefits security by presenting fewer features to verify and fewer potential attack surfaces.

HFI's $\mu$-architecture must also conform to more specific low-level constraints that are critical for practical adoption:

(1) **Low power:** Power consumption is a critical metric in small devices and datacenters — in light of this, HFI eschews the use of power-hungry components such as large caches (HFI's on-chip storage is measured in bytes, not kilobytes).

(2) **Zero impact on non-sandbox code:** Most code is not sandboxed today, and vendors are not likely to embrace a design that slows down regular (non-sandboxed) code. Thus, we cannot add extra pipeline stages that are seen by regular instructions, nor can we add timing delays to any existing stages that are on the critical timing path of the processor (and would therefore impact the CPU maximum frequency).

(3) **Low impact on circuit area:** HFI enforces region bounds checks in the neighborhood of critical pipeline structures, such as the register file, address generation unit, and the TLB. Thus, large structures, even if they are not on the critical timing path, can exacerbate timing delays between critical structures that are now further apart. For example, using 64-bit comparators to check bounds would be a natural design choice, but would challenge our power and area goals.

**Additional components:** As discussed, HFI seeks to minimize the number of new components it requires. Of course, it's not just the component count but where they are added that matters; even a handful of gates on critical paths like memory lookup would increase the cycle time of the entire CPU. To account for this, we worked with architects at a major CPU vendor to refine our design to try to minimize impact on CPU pipelines.

In total, HFI's architecture adds: 8 instructions, 22 internal 64-bit registers (10 regions specified by 2 registers each, 1 exit handler register and 1 configuration register), an additional 22 internal 64-bit registers for the optional switch-on-exit support, one 32-bit comparator for bounded regions, four 64-bit AND gates for masking, four 64-bit equality checks for prefix-checked regions, and five 2-bit muxes (region lookup, negative offset checks, etc.).

## 4.1 Implicit Regions

Implicit regions enforce memory protection for control (code regions) and data (data regions), including speculative safety.

**Data regions.** HFI supports four data regions, and allocates two registers per-region, for the `lsb_mask` and `base_prefix`. If HFI is enabled, data region checks are applied to all load and store operations on the data path (except those that employ `hmov`).

Concretely, checks work by *prefix matching*. To implement this, the region's `lsb_mask` is first used to remove the least significant bits of the effective address (with an AND operation), and then the `base_prefix` is compared for equality with the remaining bits of the address. To ensure efficiency, these checks occur in parallel with the dtb and cache index lookup.

If prefix-checking fails for all regions, or the first matched region does not have adequate permissions (e.g., no read permission for a load operation), HFI triggers a segmentation fault (using the same circuit paths that would handle an access to unmapped memory), and saves the reason for the fault in a MSR.

Since bounds checking, dtb lookup, and cache index lookups happen in parallel (as shown in Figure 1), it may appear that cache state could be modified as a result of secret (out-of-bounds) data. However, HFI is designed to prevent this sort of side-channel attack by construction: all bounds checks occur *before* the processor can resolve the physical address of a memory access. This is secure because the processor can update cache metadata like the LRU bits (for hits) or fetch new data blocks (for misses) only after resolving the physical address. HFI can therefore strictly prevent any metadata updates if there has been a fault.

Note that we cannot say the same for the dtb or the i-cache; here an out-of-bounds address can affect metadata — e.g., LRU bits. However, the invariant we guarantee — no secret (data stored outside the boundaries of the region) ever affects architectural state — is still not violated, since we do not allow the *result* of an out-of-bounds memory operation to propagate into any of these structures.

**Code regions.** Code regions enforce bounds checks on control flow (both speculative and non-speculative). HFI allocates four internal registers to store region metadata for the two code regions. HFI uses prefix-checking to bound the program counter, employing the same technique used for data regions.

To ensure security, prefix-checking is applied in parallel with the decode stage. If the check finds a matching region with execute permissions, it succeeds, and decode carries on normally. If the check fails, it prevents the decoded micro-ops from entering the pipeline, and instead translates all instructions into a faulting NOP micro-op. This ensures that instructions that are out-of-bounds are not executed during committed execution, and are also not executed speculatively.

To summarize, HFI's data pipeline is Spectre safe, since the data cache is not updated prior to bounds checks being completed; HFI's control pipeline is safe as bounds checks finish prior to instruction decode prior to the execution of instructions. This approach also helps to guarantee that any code executed as the result of PHT, BTB, and RSB (speculative) predictions are checked prior to execution.

## 4.2 Explicit Regions

Explicit regions (§3.2) offer granularity that is necessary to support Wasm heaps as well as fine-grain object sharing. They are accessed with hmov instructions, which performs bounds checks to ensure only memory specified by the region is accessed.

At a high level, the hmov instructions (hmov0, hmov1, hmov2, hmov3) follow a similar format to the standard x86 move instructions; it supports all variations and addressing modes of x86, including the complex addressing mode where *scale, index, base* and *displacement* operands are combined to form the effective address.

However, hmov has additional steps that: (1) choose an HFI region, (2) replace the *base* operand with the base address of the chosen HFI region, and (3) perform checks on the remaining operands and the resulting effective address of hmov to ensure that the memory access remains within the region. Each of these steps can be implemented with small modifications to the existing x86 data path:

**Instruction decode.** The decode pipeline stage is responsible for translating x86 instructions into simpler and easier to execute operations called micro-ops. HFI extends the decode stage with a new micro-op that differs slightly from the load or store micro-op by adding a region number and a single bit indicating that this is an hmov rather than a standard mov.

**Register read.** As shown in Figure 1, during the register read stage, hmov substitutes the base register (which would otherwise be read from the register file) with a base value read from one of the four sets of range registers.

**Bounds checking on hmov.** During memory operations, HFI performs bounds checking in parallel with the processor's address translation (dtb lookup). In the case of hmov, the HFI comparator unit (Figure 1), ensures that the effective address is within bounds.

This could be naively accomplished with two (expensive) 64-bit comparators. HFI, however, exploits the fact that the base has been precomputed — and uses this value along with three cheaper checks that require only a single 32-bit compare. Specifically, HFI checks that: (1) the 32 most significant bits of the effective address is smaller than the upper bound specified in the HFI region metadata registers, (2) the displacement and index sign bits are non-negative, and (3) effective address calculation does not cause an overflow. The second and third check ensure it is impossible to generate an effective address lower than the base. Thus, we check both base and bound with a single compare (and three trivial bit checks). Next, we discuss why checking only 32 most significant bits is secure for the two types of explicit regions HFI supports — large and small (§3.2).

**Bounds checking large and small regions.** Large regions require the base and bounds to be aligned to 64K ($2^{16}$). This means that the bottom 16-bits of the effective address can be safely ignored, as their contents cannot cause an out-of-bounds access. Additionally, x86 CPUs typically support a 48-bit virtual address space; HFI can thus ignore the top 16 bits of our address for comparison. On Intel server CPUs that support 52/57-bit address spaces, a larger 36/41-bit comparator would be necessary.

Small regions support arbitrary bounds up to 32-bits (4 GiB) in size, as long as the small region does not cross a 4 GiB boundary. Because of these restrictions any addresses in small regions cannot affect the top 32-bits of the effective address. HFI thus only checks the bottom 32-bits of the effective address of small regions.

## 4.3 Region Updates

Region registers state can be modified by several instructions: hfi_set_region, hfi_clear_region, and hfi_get_region, either when HFI is disabled — or while HFI is enabled in a hybrid sandbox. While the semantics of these operations are quite simple (e.g., writing metadata to a region register), there are several nuances to how they are implemented to ensure performance and safety.

To start, these operations do not serialize when not in HFI mode, as they are always followed by an hfi_enter (that can be serialized) before the HFI region checks take effect. However, they do serialize when executed in a hybrid sandbox, to ensure the correctness of in-flight instructions and memory operations. Additionally, hfi_set_region(code,...) flushes any pending memory operations, and is serialized to ensure that all in-flight instructions are retired prior to applying the new bounds. These serialization costs can be avoided if we employ register renaming on HFI metadata registers (similar to that used in general-purpose registers), in essence, trading complexity for improved performance.

## 4.4 Enters, Exits, Sandbox Types, & System Calls

As discussed in §3.3, HFI provides hfi_enter and hfi_exit instructions to enter and leave a sandbox. On sandbox entry, the hfi_enter instruction saves its parameters — the exit handler and flags (switch-on-exit, is-serialized, is-hybrid) to internal configuration registers, and HFI mode is enabled. On exit, HFI disables the sandboxing, records the reason for the exit (e.g., executed an hfi_exit instruction, executed a syscall, traps) in an MSR, and finally jumps to an exit handler if one is specified. The semantics of sandbox entry with respect to serialization are described in §3.4.

**Sandbox types.** Hybrid and native sandboxes differ in how they deal with region updates, privileged instructions (system calls), and the semantics of hfi_exit. However, there is no special component that implements a sandbox type. Rather, other features modify their behavior based on the sandbox type as discussed in §3.

**System call interposition.** Native sandboxes require HFI to redirect syscalls when executing sandboxed code. To do this, HFI modifies the decode stage of the syscall instruction (and all variations including sysenter, int 0x80, etc.) to first perform a microcode check if HFI is in native mode (is-hybrid is false), and redirect control flow to the HFI exit handler if this is the case. The syscall instruction is otherwise unmodified. While this approach imposes a single cycle penalty on syscall instructions for the additional check, this overhead is unlikely to impact application performance due to the relatively sparse nature of system call invocations compared to other instructions such as loads and stores.

### 4.5 Mitigating Spectre

HFI's enter, exit, and region update instructions can be serialized when necessary for Spectre protections. However, HFI also offers another way to mitigate Spectre, that minimizes serialization overhead for common use cases. As discussed in §3.4, the switch-on-exit flag allows speculative safety without the need to serialize on every entry and exit. This is safe because any speculatively run hfi_exits would atomically switch to the runtime's sandbox.

To support this, we extend HFI in three ways. First, we double the number of HFI metadata registers, so that we can store the metadata for two sandboxes — the runtime sandbox and the child sandbox. Second, we modify the hfi_enter instruction (when executed with the switch-on-exit) to preserve the trusted runtime's sandbox metadata (currently in the HFI registers), before loading the child sandbox's metadata. Finally, the hfi_exit instruction, upon execution with the switch-on-exit flag enabled, atomically switches back to using the trusted runtime sandbox metadata instead of disabling HFI.

These changes to hfi_enter and hfi_exit can be implemented in a straightforward manner in microcode. While this feature does add some additional cost in the form of internal registers, it allows us to support speculative safety for common use cases, while removing most of the cost of serialization.

## 5 EXPERIMENTAL METHODOLOGY

This section documents our experimental framework and approach (and experimental evaluation thereof) to achieving accurate hardware simulation of long-running benchmarks.

### 5.1 Integrating HFI into Wasm

We modify two compilers used in production applications to test the use of HFI with Wasm toolchains: Wasm2c, an ahead-of-time compiler used to run untrusted libraries in Firefox, and Wasmtime, a just-in-time compiler used in Fastly's FaaS environment.

We added HFI support to **Wasm2c**, and secured the Wasm heap by using a explicit region, accessed by hmov. We removed mprotect() calls to setup guard regions, and replaced existing heap growth code with hfi_set_region. We added hfi_enter and hfi_exit to sandbox transitions. We omit HFI support for the Wasm stack, indirect function tables, and the code section as these would not impact our performance results. However, this would be necessary for complete Spectre mitigation.

We modify **Wasmtime** to use HFI for lifecycle operations (Wasm sandbox creation and growth) to understand the benefits of HFI. Thus, we integrated hfi_enter, hfi_set_region, support but did not add hmov support for the Wasm heap (as this would not affect lifecycle costs). We use HFI to optimize Wasmtime's teardown of multiple sandboxes to more efficiently reclaim memory on sandbox exits. We discuss this in more detail below.

**Optimizing Wasmtime with HFI.** Wasmtime today deallocates or tears down sandboxes using the madvise(MADV_DONTNEED) system call, which discards an old sandbox memory and replaces it with a lazy copy-on-write mapping of the next executing sandbox's memory image. These madvise() calls can be slow as they incur a cost proportional to the size of region being discarded, and worse can even perform a TLB shootdown in concurrent environments. HFI allows us to optimize these madvise() syscalls by batching multiple such calls when discarding memories of sandboxes that are adjacent in memory. This is possible as HFI eliminates the need for large regions of guard pages between different sandbox memories. By eliminating these guard pages, we can trivially run madvise() across immediately adjacent heaps, without paying a significant penalty when unnecessarily discarding guard pages. Furthermore, the elimination of guard pages also reduces address-space pressure, which means we can afford to wait longer prior to discarding old sandbox memories. We thus modified Wasmtime to take advantage of batching during sandbox destruction.

### 5.2 Hardware Simulation

Our approach to accurately modeling HFI on complex, long-running applications is twofold. We model all of the low-level costs of HFI using detailed cycle-accurate pipeline simulation, but can only produce results for relatively short-running applications. Thus, we supplement this approach with a faster software emulation model and validate the correlation between the two on some representative, small benchmarks that stress key features of HFI.

**Cycle accurate pipeline model.** We configure the gem5 simulator [46] to resemble the Intel Skylake CPU on which we natively run the majority of our benchmarks. To support HFI, we add metadata registers and instructions. We add the new hmov instruction by using a new prefix for x86's mov, and we modify the syscall instruction's microcode to invoke sandbox state handlers. Further details about the simulator's implementation are in the appendix.

**Software emulation.** The gem5 simulator is several thousand times slower than native execution. We thus primarily gauge HFI's performance by emulating the cost of HFI using available instructions, described in detail in the appendix. All benchmarks with a single exception (described below) are run on an Intel Core i7-6700K (with Skylake architecture) (4 GHz) with 64 GiB of RAM,

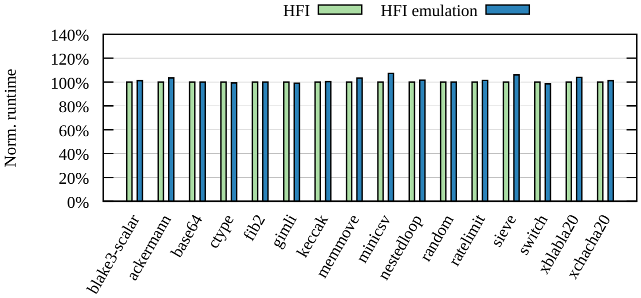


**Figure 2: Accuracy of emulated HFI. We ran our hardware simulated HFI and software emulated HFI side-by-side on the Sightglass benchmarks in gem5. We see that the emulation offers reasonable accuracy — with overheads ranging from 98%–108% of simulated overhead.**

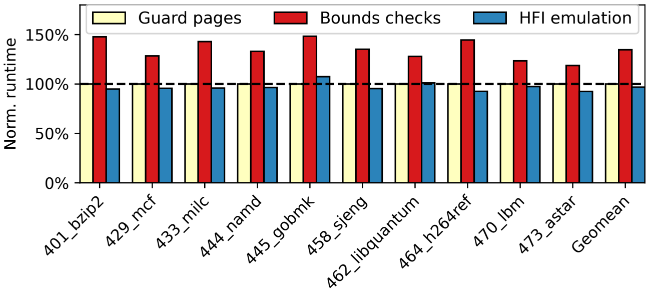


**Figure 3: SPEC INT 2006 results normalized against guard pages. Bounds-checking incurs overheads between** 18.74% **and** 48.34%**, with median and geometric mean** 34.67%**. On the other hand, HFI takes between** 92.51% **and** 107.45% **the execution time of guard pages, with median** 95.88% **(a speedup of** 4.3%**) and geometric mean** 96.85% **(a speedup of** 3.25%**).**

running Ubuntu 20.04.5 LTS. Our benchmark measuring the performance overheads of HFI's native sandbox (§6.4.2) is run on an Intel Core i7-1165G7 (with Tigerlake architecture) (2.80 GHz) with 16 GiB of RAM, running Ubuntu 22.04.1 LTS; the second machine supports Intel MPK and allows us to contrast its performance with HFI. Benchmarks are run with both CPU frequency scaling and hyperthreading disabled. We also pin benchmarks to a single CPU that is isolated from other processes with CPU shield. We record execution time on benchmarks using the Hyperfine timing utility, which accounts for warmup runs and averages across several subsequent runs. For the hybrid sandbox (with Wasm) evaluations, we compile source files using stock wasi-clang (i.e., clang with Wasm as a target) and Wasm2c set to employ different protection mechanisms — guard pages, bounds-checking, or HFI emulation. We use a native C compiler (clang or GCC) for native sandbox evaluations.

**Cross validation.** To vet our emulation, we compare gem5 HFI simulation against our emulation of HFI performance using the Sightglass benchmark suite. Sightglass consists of various short Wasm-friendly programs, mainly primitives from cryptography, mathematics, string manipulation, and control flow. We exclude those Sightglass benchmarks which are incompatible with Wasm2c, or require over a day to execute on gem5. Figure 2 shows the comparison between HFI and its emulation. Across the suite, benchmarks in software emulation have cycle counts between 98% and 108% of the simulation. The geometric mean difference in runtime is 1.62%.

## 5.3 Security Evaluation

To ensure that out-of-bounds memory accesses trap, we employ a collection of unit tests on our HFI gem5 simulation. To ensure our simulation's Spectre resistance, we use exploits from the TransientFail [11] and Google SafeSide [24] test suites. We run the in-place Spectre-PHT attack from Google SafeSide in the gem5 simulator to demonstrate that sandboxed code can speculatively access secret data (stored in a global variable in the host application for this example) when executed without HFI. We then check that HFI prevents this attack when the host application protects this global variable using HFI's regions (the memory range containing the global variable is in an HFI region without read or write permissions). In Figure 7 (in the appendix), we plot the memory access latencies to ensure that HFI is able to prevent speculative access of secrets and the subsequent cache-based exfiltration. We similarly run the in-place Spectre-BTB attack from the TransientFail test suite, and check that this is also mitigated[7].

# 6 HFI PERFORMANCE

We integrate HFI into standard benchmarks and real-world software, and evaluate its performance. We examine four important use cases: long-running applications (SPEC), library sandboxing in a browser, a JIT-based FaaS, and native sandboxing in a server workload (NGINX). Finally, we examine HFI as a Spectre mitigation.

## 6.1 SPEC 2006 Benchmark Suite

SPEC CPU 2006 is a suite of integer and floating-point workloads. We use SPEC06 instead of SPEC17, as many of SPEC17's benchmarks require more memory than the 4 GiB that Wasm supports.

We evaluate the impact of HFI on the subset of SPEC06 that is compatible with Wasm and the WASI SDK fork of clang. Figure 3 shows the performance of bounds checking and HFI compared to guard pages. These are long-running applications that do not test HFI's fast transitions, but do show its low cost in steady state.

In our results, we see that HFI (geomean speedup of 3.2% over guard pages) is far less costly than bounds checking (geomean slowdown of 34.7% over guard pages), and HFI on average is modestly faster than guard pages. The 445.gobmk benchmark takes a little longer with HFI as it puts heavy pressure on the instruction cache, and in this case, we see that the hmov instructions for which we used longer encodings, impacts HFI performance. We emphasize, however, that HFI is the only scheme of these three that also offers Spectre protections. Securing Wasm compilers against Spectre, without HFI, incurs a 62% to 100% hit [53].

Finally, this benchmark only invokes `hfi_enter` and `hfi_exit` once each, at the beginning and end of each benchmark; thus the serialization overhead added to this benchmark is negligible.

[7] For completeness, we note that gem5 does not model BTB speculation with sufficient detail to perform a real Spectre-BTB attack. Thus, in our tests, we instead model a Spectre-BTB attack using concrete control flow that leaks secret data using the cache-side channel.

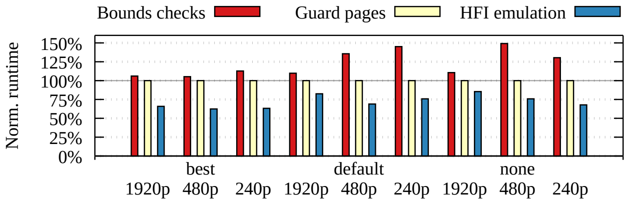


**Figure 4: Firefox image rendering. HFI offers a significant speedup for image rendering — the biggest increase for larger images that amortize the cost of `hfi_enter`. More compressed images — that are more compute intensive — also see greater benefits, as a result of decreased register pressure.**

To understand why HFI improved performance, we dug into two effects more deeply: heap growth overhead and register pressure. Heap growth is an expensive operation in a sandbox that uses guard pages, as it requires a call to `mprotect()`, while HFI can just update a region's bound registers. To measure this difference, we ran a simple benchmark in Wasmtime that grows the Wasm heap from a single page to 4 GiB in 64 KiB increments. In total, the `mprotect()` method takes 10.92 seconds, while HFI takes 370 ms, a difference of ≈ 30×, reflecting HFIs impact on heap growth after being amortized in the context of the Wasmtime runtime.

HFI also removes the need to load Wasm's memory base and bounds into general-purpose registers for software-based checks — reducing register pressure. To approximate the impact of this, we ran Wasmtime's Spidermonkey benchmark, first reserving one register, then reserving two registers. We find that reserving one register incurs an overhead of 2.25%, while reserving two registers incurs an overhead of 2.40%. Thus, we have a rough approximation of the improvements HFI can offer in this dimension.

## 6.2 Wasm Sandboxing in Firefox

To understand the end-to-end performance impact of HFI, we measure the performance of Wasm sandboxed font and image rendering in Firefox, similar to Narayan et. al [52], with and without HFI. The font rendering benchmark reflows the text on a page ten times via the sandboxed `libgraphite`, using multiple font sizes to avoid any effects from font caches. When using Wasm with guard pages, `libgraphite` renders this in 1823 ms; using bounds-checking, 2022 ms; and using HFI emulation, 1677 ms.

We also test Firefox's performance using a Wasm-sandboxed `libjpeg`. For this, we measure decode time for JPEG-format test images from the Image Compression benchmark suite. We use images of three resolutions and three compression levels. Figure 4 shows the median decode times for each configuration out of 1000 runs. As expected, HFI emulation offers the fastest sandboxing compared to the typical software-based enforcement of Wasm. HFI is faster than software bounds checks by design: at a hardware level, memory access validation happens in parallel with the memory access itself. HFI is also faster than guard pages, because it can elide calls to `mprotect()` (which is needed by guard pages) in favor of relying on hardware to enforce access safety.

In the font rendering benchmark, HFI outperforms guard pages by 8.7%. In the `libjpeg` tests, the speedup of HFI over guard pages is between 14% and 37%. Even though this benchmark has a sandbox invocation per line of pixels (a 1080x720 image requires ≈ 720 × 2 serialized enters/exits), Figure 4 shows that even with this added overhead, HFI's performance benefits are able to amortize this cost.

## 6.3 HFI's Impact on Wasm-based FaaS Platforms

One of the primary uses of Wasm (outside the browser) is to isolate tenants from one another on FaaS platforms like Fastly's Compute@Edge [20] or Cloudflare Workers [76]. We evaluate the performance and scaling impact of using HFI for such systems.

*6.3.1 Cost of Sandbox Setup and Teardown.* One of the bottlenecks in FaaS platforms is the setup and teardown of sandboxes. §5.1 describes how HFI allows us to coalesce many sandbox teardowns into one large teardown. To evaluate the impact of this optimization, we use a custom FaaS benchmark that creates 2000 sandboxes, executes a trivial short-lived workload on each (writes some constant data to the sandbox's memory) and then tears down the sandboxes.

We run this on three versions of Wasmtime: (1) stock Wasmtime that invokes `madvise()` once per sandbox on teardown; (2) HFI-wasmtime that batches `madvise()` teardowns; and (3) non-HFI Wasmtime that batches `madvise()`, but does so without eliding guard pages. We find that stock Wasmtime has a per-sandbox teardown cost of 25.7 $\mu$s, HFI-Wasmtime took 23.1 $\mu$s (a 10.1% improvement), and non-HFI batched teardown took 31.1 $\mu$s. Thus coalescing calls to `madvise()` during teardown improves performance, but only when using HFI, as it lets the runtime elide guard pages.

*6.3.2 Scalability of Sandbox Creation.* HFI's ability to let Wasm runtimes elide guard pages also impacts scalability, i.e., the number of sandboxes that can be concurrently executed by Wasm runtimes. We test this by measuring the number of 1 GiB Wasm sandboxes that can be created by Wasmtime when it is allowed to elide guard pages (by using HFI). When eliding guard pages, we find that Wasmtime can create up to 256,000 1 GiB sandboxes in a single process, i.e., the application can make full use of its address space. This 256,000 limit would be even larger for smaller-sized sandboxes.

## 6.4 Performance of HFI's Native Sandbox

We also examine HFI's native sandbox that sandboxes code without recompilation. This has two costs — trapping syscalls and switching protection domains. We evaluate these below.

*6.4.1 Performance of Trapping Syscalls.* Sandboxing systems that isolate native code restrict its access to system resources by interposing on system calls. State-of-the-art native code isolation systems like ERIM [75] rely on Seccomp-bpf filters for this, whereas HFI has direct microarchitectural support for system call interposition. To compare the overhead of these two techniques, we ran a custom syscall benchmark that opens a file, reads it, and closes it 100,000 times, and uses Seccomp-bpf and HFI in turn to interpose on the syscalls. We found that using Seccomp-bpf version imposes an overhead of 2.1%, over the HFI version.

*6.4.2 Switching Costs of Sandboxing OpenSSL in NGINX.* We modify the NGINX webserver to estimate the performance of sandboxing crypto functions and session keys in OpenSSL, similar to ERIM [75]. We use this to measure the costs of integrating HFI

**Table 1: Impact of HFI Spectre protection on tail latency. We compared HFI and Swivel — the fastest software-based Spectre mitigation, on several Wasm FaaS workloads. Swivel increased tail latency by 9%–42%. HFI's increased tail latency by 0%–2%.**

| **HFI Protection** | XML to JSON | | | | Image classification | | | | Check SHA-256 | | | | Templated HTML | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Avg Lat | Tail Lat | Thru-put | Bin size | Avg Lat | Tail Lat | Thru-put | Bin size | Avg Lat | Tail Lat | Thru-put | Bin size | Avg Lat | Tail Lat | Thru-put | Bin size |
| Lucet(Unsafe) | 421 ms | 466 ms | 231 | 3.5 MiB | 12.2 s | 14.7 s | 1.62 | 34.3 MiB | 589 ms | 667 ms | 161 | 3.9 MiB | 45.6 ms | 61.8 ms | 2.19k | 3.6 MiB |
| Lucet+HFI | 431 ms | 480 ms | 227 | 3.5 MiB | 12.2 s | 14.7 s | 1.62 | 34.3 MiB | 602 ms | 647 ms | 165 | 3.9 MiB | 45.7 ms | 61.2 ms | 2.18k | 3.6 MiB |
| Lucet+Swivel | 559 ms | 616 ms | 174 | 4.1 MiB | 11.5 s | 12.8 s | 1.72 | 34.5 MiB | 645 ms | 709 ms | 150 | 4.6 MiB | 78.9 ms | 97.9 ms | 1.26k | 4.2 MiB |

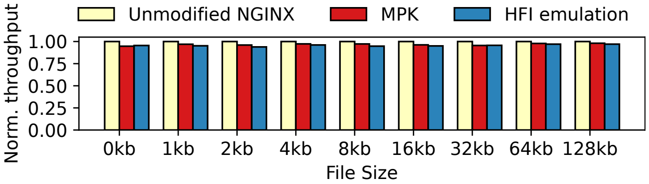


**Figure 5: Overhead of the native sandbox. We emulated the overhead of sandboxing OpenSSL in NGINX with HFI's native sandbox. As native sandbox imposes no runtime overhead, we are seeing the impact on the pipeline of serializing `hfi_enter` and `hfi_exit`. HFI's overhead is slightly larger than MPK based sandboxes [75], as HFI must additionally move region metadata from memory to registers.**

in existing applications versus the benefits (e.g., blocking attacks like Heartbleed [29] and Spectre). In particular, NGINX switches in-to and out-of sandboxed code regions rapidly, when a web client accesses encrypted content. Thus, we use this to understand the impact of serialization added by `hfi_enter` and `hfi_exit`.

HFI's native sandbox by design does not impose any execution overhead, as there is no modification of the instruction stream and region checks execute in parallel with address translation. Instead, overheads only appear during sandbox enters and exits, metadata manipulation (e.g., `hfi_set_metadata`), and traps.

Figure 5 compares the throughput of the unmodified NGINX web server delivering content with unprotected session keys versus the throughput when protecting session keys with HFI and MPK respectively. Following the experimental setup of ERIM [75], we used the *apache-bench* to send millions of requests of various sizes to the NGINX web server running on a single isolated CPU core. For each file size, we sent 2,000,000 session requests from 100 clients for 60 seconds and measured the throughput. We observe that HFI's native sandbox has a low overhead that ranges from 2.9% to 6.1%. HFI's overhead is slightly larger than MPK-based protections, which range from 1.9% to 5.3%. This is because HFI takes a few cycles to move metadata from memory to HFI registers on each transition.

## 6.5 Cost of Spectre Protections

We compared HFI's Spectre protections, to the performance of Swivel [53], the fastest known compiler-based approach mitigating Spectre in Wasm. We evaluated this by running several common Wasm workloads in the Rocket webserver and securing these workloads against Spectre. We compared Rocket with: the Lucet Wasm compiler without Spectre protections, Lucet+Swivel-SFI protections, and with Lucet+HFI using native sandbox. We also record the workload binaries' sizes.

Table 1 shows that HFI guards against Spectre with very low drop in tail-latency and no noticeable binary bloat, while Swivel incurs noticeable overheads for the same. In fact, the only overheads imposed by native sandbox HFI are due to region construction and sandbox state transitions (two per connection), and these costs are amortized by the cost of the workload.

# 7 RELATED WORK

In the last few decades, diverse approaches to fine grain isolation have been explored, through software-based, OS-based, and hardware-based techniques.

**Software-based isolation.** SFI systems [78, 86] such as Wasm [28] offer software-enforced isolation leveraging compilers to restrict memory accesses to a linear region of memory, restrict control transfers, and interpose on system calls. As discussed in §2, this approach has broad compatibility with existing code due to its simplicity, but incurs both a substantial virtual memory footprint due to guard pages and performance overheads when trying to isolate JIT code [5] or prevent Spectre attacks [53, 67] in software. Systems that isolate computations using safe languages such as Singularity [19] or Erlang [6] offer fast context switching and good scalability, but cannot isolate existing code written in systems languages like C or C++. HFI is not intended to replace these systems, but instead complements these software-based approaches with hardware primitives for performant Spectre-safe isolation.

**OS-based isolation.** OS operating systems, most notably microkernels such as L4 [31] have long pushed the boundaries of isolation with page-based protection hardware. Systems have experimented with offering process-style isolation primitives to an application through the interface of a "sandboxed thread" [13, 33]. Lightweight contexts [43] go further and allow an existing application thread to switch between sandboxed contexts. These systems have the benefit of not relying on hardware changes, but have performance overheads due to expensive protection ring crossings (calls into the OS kernel) to switch the active sandbox, as well as restrictions on scaling due to limits on the total hardware-supported process contexts (ASIDs) — 4096 on Intel CPUs [35].

**Hardware-assisted isolation with page metadata.** Diverse systems have proposed extensions to page table metadata to store a per-sandbox ID checked by hardware, for example Donky-x86 [66] and others [22], or have cleverly reused the existing checked page-level metadata (for x86 rings [40], virtual machines [9, 25, 44, 60, 81], ARM's memory domains [89], trusted execution environments [7, 8]) to provide isolation at a page-level. Unlike HFI, these approaches do not require a sandboxes' memory to be in a contiguous range. These approaches, however, inherit the limitations of page-based isolation (see Norton [54]). For example, they require expensive calls to kernel code to update the page-level metadata (to grow the sandbox-accessible memory) or switch the active sandbox; worse, keeping this metadata consistent across CPU cores often becomes a performance bottleneck, due to the need for expensive TLB clears or shootdowns. Such systems also typically rely on separate tools to interpose on syscalls that may break isolation [14, 34, 58, 65, 66, 77], and are incompatible with zero-cost transitions [38] (fast sandbox entries/exits that do not have to save and restore registers) since they do not distinguish between stack and heap protection.

**Intel MPK**, while largely similar to the above approaches, allows switching of the active MPK domain (the current sandbox) in userspace (i.e., ring 3). MPK-based techniques [30, 34, 70, 75, 80] have thus been explored to reduce context switch overheads and sandbox native binaries, however these tools still face the other limitations of page-based approaches described above. MPK also only supports 15 domains/sandboxes efficiently; thus, making it unsuitable for server-side applications, which handle many thousands of unique requests or even client-side applications that require dozens or hundreds of unique contexts [52]. Techniques that attempt to scale MPK domains do so by falling back on other page-level techniques (e.g., page permissions [57], virtualization [27]) incurring their associated performance costs.

**Hardware-assisted isolation with range checks.** Isolation systems have also been built around existing hardware such as x86 Segmentation [21, 86] and Intel MPX [39] which rely on explicit bounds checks on addresses in memory operations.

Segmented memory architectures are some of the oldest protection mechanisms [42], and have been used by VMMs [1], by operating systems like Multics [62], and for both OS components and applications in systems like OS/2 and AS400. Earlier SFI implementations such as Vx32 [21] and NaCl [86] leveraged **x86 segmentation** on 32-bit platforms for fast isolation — an approach that has similarities to HFI. Unfortunately, x86-64 dropped support for segmentation, thus this technique has limited utility on current hardware. HFI offers some primitives that are similar to x86 segmentation (for example the segment/region relative addressing of explicit regions) but pairs this with primitives adapted to the flat memory model that can isolate unmodified applications (implicit regions). Additionally, HFI's abstractions and implementation are much more minimal and tailored to in-process isolation, avoiding complex segmentation features such as call-gates or automatically switching privilege-levels when changing regions/segments.

**Intel MPX** (Memory Protection Extensions), a hardware feature to support fine-grain memory safety, has also been re-purposed by systems like Memsentry [39] for sandboxing. While this approach would work well in theory, practical implementations incurred substantial overheads comparable to software-based sandboxing, in part due to the data dependencies created by multiple range checks prior to memory accesses [39]. HFI, in contrast, carefully avoids multiple range checks by design, and instead relies on a single range check for explicit regions, and masking for implicit regions. HFI also accounts for other requirements of isolation systems such as system call interposition and Spectre resistance.

Newer efforts like the **J-extension** [47] on RISC-V have proposed hardware for address masking (similar to the software-based approach of Wahbe et al. [78]); however, this inherits all the drawbacks of software-based masking such as the converting of out-of-bounds memory accesses into random memory corruption (§2).

**Hardware-assisted isolation with capabilities.** Capability-based addressing has a long history [12, 17, 18, 51, 74], and is seen most recently in **CHERI** [84]. CHERI provides a powerful security model that can represent an unlimited number of byte-granular protected regions, which enables not just compartmentalization but also object-level memory safety. However, it also requires extensive modification to nearly every layer of the software and hardware stack: including the OS [73], ABI [16], and compiler [72] and extensive hardware support. CHERI's use of 128-bit fat pointers to track capability metadata also comes at the cost of increased memory and cache footprint that affects performance [83]. Alternative capability systems leveraging cryptography [41] have also been proposed. In contrast, HFI takes a far more minimalist approach, focusing exclusively on supporting SFI and in-process sandboxing, and thus only requires small modifications to existing processors, and little to no change to existing software.

**Hardware changes for Spectre-resistant CPUs.** Several works like speculative taint tracking [87] and others [45, 82, 85] have proposed redesigning the CPU's approach to speculative execution to prevent Spectre attacks. These approaches are complementary to HFI in that they provide Spectre protection for general programs, i.e., they provide protection for programs that do not run sandboxed code, but do so at the cost of adding complexity to the CPU design. In contrast, HFI is designed to provide isolation and Spectre safety for sandboxing and is explicitly designed to minimize changes to the CPU to allow easy adoption.

## 8 CONCLUSION

Modern page-based protection architectures are remarkably powerful. They have served us for decades, during massive changes in processor speeds, compute form factors, and workloads. However, one thing they are not good at is the kind of fine-grain isolation where software excels. This lack of support for fine-grain isolation is a problem and limits the way we think about building systems.

Wasm has opened the door to a new world of in-process sandboxing, safe extensibility, and high-scale concurrency. However, being software-based also brings a host of constraints that limit its potential.

HFI offers a path beyond these constraints by supporting in-process isolation for Wasm and native binaries that preserves the benefits of software-based isolation: low context-switch overheads, fast instantiation, etc. while offering a new level of safety, scalability, and performance.

## ACKNOWLEDGMENT

Thanks to Dan Gohman and Luke Wagner from Fastly and the architects from Intel for their insightful discussions, the anonymous reviewers and shepherd for their valuable comments for improving the quality of this paper. This work was supported in part by a Sloan Research Fellowship; by the NSF under Grant Numbers CNS-2155235, CNS-2120642, and CAREER CNS-2048262; by gifts from Intel, Google, and Mozilla; and by DARPA HARDEN under contract N66001-22-9-4017. And finally, thanks to our families, without whose support this work would not be possible.

# A APPENDIX

## A.1 HFI Software Interface

```
sandbox_t:
  is_hybrid: bool        # use the hybrid sandbox
  is_serialized: bool    # serialize enter/exit
  switch_on_exit: bool   # use switch-on-exit extension
  exit_handler: u64      # if set, interpose on hfi_exit
                     #(and syscalls in native sandboxes)

hfi_enter(sandbox_t)     # enter a sandbox with params
hfi_reenter() # renter the sandbox that was just exited
hfi_exit()               # exit the sandbox

implicit_code_region_t:
  base_prefix: u64       # base address prefix
  lsb_mask: u64          # mask for address suffix
  permission_exec: bool  # execute permission

implicit_data_region_t:
  base_prefix: u64       # base address prefix
  lsb_mask: u64          # mask for address suffix
  permission_read: bool  # read permission
  permission_write: bool # write permission

explicit_data_region_t:
  # Large regions: base, bound are multiples of 64k.
  # Small regions: region shouldn't span a 4GiB boundary
  base_address: u64
  bound: u64
  permission_read: bool  # read permission
  permission_write: bool # write permission
  is_large_region: bool  # use large/small region

region_t = { implicit_code_region_t
           | implicit_data_region_t
           | explicit_data_region_t}

# (0-1) code, (2-5) implicit_data, (6-10) explicit_data
region_number: u8

hfi_clear_all_regions()  # clear all HFI registers
hfi_clear_region(region_number) # clear a given region
# set or get HFI registers of a region
hfi_set_region(region_number, region_t *)
hfi_get_region(region_number, region_t *)
```

**Figure 6: The HFI interface. The functions represent HFI instructions, while the structures represent the parameters passed in to the HFI instructions.**

## A.2 HFI Hardware Simulation

**Gem5 simulation.** We create a gem5 instance like table 2.

(1) `hfi_set_region`: We move data from memory into new hardware registers we add to gem5 to store HFI metadata. This metadata includes information such as region masks, the exit handler etc.

(2) `hfi_enter` and `hfi_exit`: These instructions write to the `enforcement` bit register to enable/disable sandboxing. We serialize the pipeline upon these instructions. The `hfi_exit` additionally jumps to a exit handler if specified.

(3) `hmov`: We add `hmov` as a variant of the x86 `mov` instruction which takes a prefix. The data and code pipeline masks on `mov` instructions and program counter are implemented to avoid any delays.

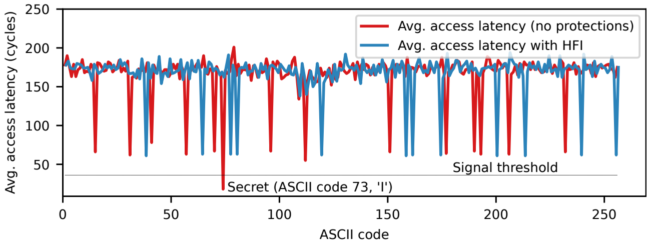


**Figure 7: Access latencies in the SafeSide [24] Spectre POC. SafeSide attacks rely on exfiltrating speculatively accessed data via the cache; thus data access latencies indicate whether secret data has been accessed. Without HFI, we see a clear signal (low access latency), corresponding to accessing the first byte of the secret (the letter 'I') in the SafeSide POC. In contrast, with HFI, we don't see access latencies that is below the measured threshold of the Spectre attack.**

(4) `syscall` microcode: We modify the microcode of privilege-escalating instructions like `syscall` to invoke the HFI handler (if specified) while inside an HFI sandbox.

**Table 2: Architecture detail for the baseline x86 core**

| Baseline Processor | | | |
|---|---|---|---|
| Frequency | 3.3 GHz | i-cache | 32 KiB, 8-way |
| Fetch Width | 16 B | d-cache | 32 KiB, 8-way |
| Issue Width | 8 uops | Decode Width | 5 uops |
| INT/FP Regfile | 186/144 regs | IQ | 97 entries |
| LQ/SQ Size | 64/36 entries | Functional Units | Int ALU(4), Mult(1), FPALU/Mult(2) |
| ROB Size | 224 entries | | |

**Software emulation.** Overhead emulation using currently available instructions.

(1) `hfi_set_region`: We emulate its cost by moving the hfi region metadata from memory to general-purpose registers.

(2) `hfi_enter` and `hfi_exit`: We use `cpuid`—an instruction known to serialize the pipeline, to capture their serialization cost [35]. `hfi_exit` also checks to see if a syscall handler is specified to capture the cost that would normally occur in the microcode implementation.

(3) `hmov`: We emulated it with a regular `mov` instruction that uses a fixed region with a base of `0x7ffff000`, i.e., one page prior to 2 GiB. This is the largest page-aligned address the x86 `mov` instruction can refer to via its constant field (without a register). We use this to emulate costs as (1) it reserves one of the inputs to the x86 `mov` consistent with `hmov` and (2) it captures the host program's speedups due to not having to use a general-purpose register to store the base.

This `hmov` emulation choice is deliberate as this is the largest address x86 can address, *while* still allowing compilers to provide two arguments to the mov instruction. It also captures the HFI benefit of reduced register pressure as it allows the host program to not consume a register for the region base.

## REFERENCES

[1] Ole Agesen, Alex Garthwaite, Jeffrey Sheldon, and Pratap Subrahmanyam. 2010. The evolution of an x86 virtual machine monitor. *ACM SIGOPS Operating Systems Review* 44, 4 (2010), 3–18. https://doi.org/10.1145/1899928.1899930

[2] Akamai. 2015. Serverless Computing with Akamai Edge Workers. https://www.akamai.com/products/serverless-computing-edgeworkers.

[3] Andreas Rossberg (Ed.). 2020. Memory64 Proposal for WebAssembly. https://github.com/WebAssembly/memory64.

[4] Andreas Rossberg (Ed.). 2022. Multi Memory Proposal for WebAssembly. https://github.com/WebAssembly/multi-memory.

[5] Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff L. Biffle, and Bennet Yee. 2011. Language-Independent Sandboxing of Just-in-Time Compilation and Self-Modifying Code. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. https://doi.org/10.1145/1993498.1993540

[6] Joe Armstrong. 2010. erlang. *Commun. ACM* 53, 9 (2010), 68–75. https://doi.org/10.1145/1810891.1810910

[7] Sergei Arnautov, Bohdan Trach, Franz Gregor, Thomas Knauth, Andre Martin, Christian Priebe, Joshua Lind, Divya Muthukumaran, Dan O'Keeffe, Mark L. Stillwell, David Goltzsche, David Eyers, Rüdiger Kapitza, Peter Pietzuch, and Christof Fetzer. 2016. SCONE: Secure Linux Containers with Intel SGX *(OSDI'16)*. USENIX Association, USA, 689–703.

[8] Ahmed M Azab, Peng Ning, Jitesh Shah, Quan Chen, Rohan Bhutkar, Guruprasad Ganesh, Jia Ma, and Wenbo Shen. 2014. Hypervision across worlds: Real-time kernel protection from the arm trustzone secure world. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. https://doi.org/10.1145/2660267.2660350

[9] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-level Access to Privileged CPU Features. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[10] Bugzilla Bug 1791598 2022. Evaluate expat CVE-2022-40674 fix. https://bugzilla.mozilla.org/show_bug.cgi?id=1791598.

[11] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. 2019. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *Proceedings of the USENIX Security Symposium (USENIX Security)*.

[12] Nicholas P Carter, Stephen W Keckler, and William J Dally. 1994. Hardware support for fast capability-based addressing. *ACM SIGOPS Operating Systems Review* 28, 5 (1994). https://doi.org/10.1145/381792.195579

[13] Yaohui Chen, Sebassujeen Reymondjohnson, Zhichuang Sun, and Long Lu. 2016. Shreds: Fine-grained execution units with private memory. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. https://doi.org/10.1109/SP.2016.12

[14] R Joseph Connor, Tyler McDaniel, Jared M Smith, and Max Schuchard. 2020. PKU Pitfalls: Attacks on PKU-based Memory Isolation Systems. In *Proceedings of the USENIX Security Symposium (USENIX Security)*.

[15] Victor Costan and Srinivas Devadas. 2016. Intel SGX explained. *Cryptology ePrint Archive* (2016).

[16] Brooks Davis, Robert N. M. Watson, Alexander Richardson, Peter G. Neumann, Simon W. Moore, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Wesley Filardo, Khilan Gudka, Alexandre Joannou, Ben Laurie, A. Theodore Markettos, J. Edward Maste, Alfredo Mazzinghi, Edward Tomasz Napierala, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, and Jonathan Woodruff. 2019. CheriABI: Enforcing Valid Pointer Provenance and Minimizing Pointer Privilege in the POSIX C Run-Time Environment. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. Association for Computing Machinery, New York, NY, USA, 379–393. https://doi.org/10.1145/3297858.3304042

[17] Jack B. Dennis and Earl C. Van Horn. 1965. Programming semantics for multiprogrammed computations. In *Proceedings of the ACM Programming Languages and Pragmatics Conference*, Vol. 26. https://doi.org/10.1145/357980.357993

[18] R. S. Fabry. 1974. Capability-Based Addressing. *Commun. ACM* 17, 7 (jul 1974), 403–412. https://doi.org/10.1145/361011.361070

[19] Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R Larus, and Steven Levi. 2006. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. https://doi.org/10.1145/1217935.1217953

[20] Adam Foltzer. 2019. The Lifecycle and Performance of a Lucet Instance. https://www.fastly.com/blog/lucet-performance-and-lifecycle. Accessed: 2022-08-10.

[21] Bryan Ford and Russ Cox. 2008. Vx32: Lightweight User-level Sandboxing on the x86. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[22] 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)*.

[23] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. 2020. Sledge: A Serverless-First, Light-Weight Wasm Runtime for the Edge. In *Proceedings of the International Middleware Conference (Middleware)*. https://doi.org/10.1145/3423211.3425680

[24] Google. 2020. SafeSide. https://github.com/google/safeside.

[25] Nuwan Goonasekera, William Caelli, and Colin Fidge. 2015. LibVM: an architecture for shared library sandboxing. *Software: Practice and Experience* 45, 12 (2015). https://doi.org/10.1002/spe.2294

[26] W3C Webassembly Community Group. 2023. Component Model design and specification. https://github.com/webassembly/component-model. Accessed: 2022-08-20.

[27] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. 2022. EPK: Scalable and Efficient Memory Protection Keys. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[28] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation (PLDI)*. https://doi.org/10.1145/3062341.3062363

[29] Heartbleed 2014. CVE-2014-0160 Detail. https://nvd.nist.gov/vuln/detail/CVE-2014-0160.

[30] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L. Scott, Kai Shen, and Mike Marty. 2019. Hodor: Intra-Process Isolation for High-Throughput Data Plane Libraries. In *Proceedings of the USENIX Annual Technical Conference (ATC)*. http://www.usenix.org/conference/atc19/presentation/hedayati-hodor

[31] Gernot Heiser and Kevin Elphinstone. 2016. L4 Microkernels: The Lessons from 20 Years of Research and Deployment. *ACM Trans. Comput. Syst.* 34, 1, Article 1 (2016), 29 pages. https://doi.org/10.1145/2893177

[32] Pat Hickey. 2019. Announcing Lucet: Fastly's native WebAssembly compiler and runtime. https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime.

[33] Terry Ching-Hsiang Hsu, Kevin Hoffman, Patrick Eugster, and Mathias Payer. 2016. Enforcing Least Privilege Memory Views for Multithreaded Applications. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. https://doi.org/10.1145/2976749.2978327

[34] Bumjin Im, Fangfei Yang, Chia-Che Tsai, Michael LeMay, Anjo Vahldiek-Oberwagner, and Nathan Dautenhahn. 2021. The Endokernel: Fast, Secure, and Programmable Subprocess Virtualization. https://doi.org/10.48550/ARXIV.2108.03705

[35] Intel 2020. Intel® 64 and IA-32 Architectures Software Developer's Manual.

[36] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[37] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. 2021. Доверя́й, но проверя́й: SFI safety for native-compiled Wasm. In *Proceedings of the Network and Distributed Systems Security Symposium (NDSS)*. https://doi.org/10.14722/ndss.2021.24078

[38] Matthew Kolosick, Shravan Narayan, Conrad Watt, Michael LeMay, Deepak Garg, Ranjit Jhala, and Deian Stefan. 2022. Isolation Without Taxation: Near Zero Cost Transitions for SFI. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. https://doi.org/10.1145/3498688

[39] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No Need to Hide: Protecting Safe Regions on Commodity Hardware. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. https://doi.org/10.1145/3064176.3064217

[40] Hojoon Lee, Chihyun Song, and Brent Byunghoon Kang. 2018. Lord of the x86 rings: A portable user mode privilege separation architecture on x86. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. https://doi.org/10.1145/3243734.3243748

[41] Michael LeMay, Joydeep Rakshit, Sergej Deutsch, David M. Durham, Santosh Ghosh, Anant Nori, Jayesh Gaur, Andrew Weiler, Salmin Sultana, Karanvir Grewal, and Sreenivas Subramoney. 2021. Cryptographic Capability Computing. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1145/3466752.3480076

[42] Henry M. Levy. 1984. *Capability-based Computer Systems*. Digital Press.

[43] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation (OSDI)*.

[44] Yutao Liu, Tianyu Zhou, Kexin Chen, Haibo Chen, and Yubin Xia. 2015. Thwarting Memory Disclosure with Efficient Hypervisor-Enforced Intra-Domain Isolation. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. https://doi.org/10.1145/2810103.2813690

[45] Kevin Loughlin, Ian Neal, Jiacheng Ma, Elisa Tsai, Ofir Weisse, Satish Narayanasamy, and Baris Kasikci. 2021. DOLMA: Securing Speculation with the Principle of Transient Non-Observability. In *Proceedings of the USENIX Security Symposium (USENIX Security)*.

[46] Jason Lowe-Power, Abdul Mutaal Ahmad, Ayaz Akram, Mohammad Alian, Rico Amslinger, Matteo Andreozzi, Adrià Armejach, Nils Asmussen, Brad Beckmann, Srikant Bharadwaj, Gabe Black, Gedare Bloom, Bobby R. Bruce, Daniel Rodrigues Carvalho, Jeronimo Castrillon, Lizhong Chen, Nicolas Derumigny, Stephan Diestelhorst, Wendy Elsasser, Carlos Escuin, Marjan Fariborz, Amin Farmahini-Farahani, Pouya Fotouhi, Ryan Gambord, Jayneel Gandhi, Dibakar Gope, Thomas Grass, Anthony Gutierrez, Bagus Hanindhito, Andreas Hansson, Swapnil Haria, et al. 2020. The gem5 Simulator: Version 20.0+. https://doi.org/10.48550/ARXIV.2007.03152

[47] Martin Maas. 2022. Working Draft of the RISC-V J Extension Specification. https://github.com/riscv/riscv-j-extension.

[48] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles (OSDI)*. https://doi.org/10.1145/3132747.3132763

[49] Jordon Mears. 2019. How we're bringing Google Earth to the web. https://web.dev/earth-webassembly/.

[50] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: Better, Faster, Stronger SFI for the x86. In *Proceedings of the ACM SIGPLAN Conference on Programming language design and implementation (PLDI)*. https://doi.org/10.1145/2254064.2254111

[51] G. J. Myers and B. R. S. Buckingham. 1980. A Hardware Implementation of Capability-Based Addressing. *SIGOPS Oper. Syst. Rev.* 14, 4 (oct 1980), 13–25. https://doi.org/10.1145/850708.850709

[52] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *Proceedings of the USENIX Security Symposium (USENIX Security)*.

[53] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, and Deian Stefan. 2021. Swivel: Hardening WebAssembly against Spectre. In *Proceedings of the USENIX Security Symposium (USENIX Security)*.

[54] Robert M Norton. 2016. *Hardware support for compartmentalisation*. Technical Report. University of Cambridge, Computer Laboratory.

[55] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2018. Intel MPX Explained: A Cross-Layer Analysis of the Intel MPX System Stack. *SIGMETRICS Perform. Eval. Rev.* 46, 1 (jun 2018), 111–112. https://doi.org/10.1145/3292040.3219662

[56] Warren Pamukoff. 2022. Shopify Functions Unlocks Backend Logic to Help Meet Any Business Need. https://www.shopify.com/nz/partners/blog/shopify-functions.

[57] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. 2019. libmpk: Software Abstraction for Intel Memory Protection Keys (Intel MPK). In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[58] Dinglang Peng, Congyu Liu, Tapti Palit, Pedro Fonseca, Anjo Vahldiek-Oberwagner, and Mona Vij. 2023. uSWITCH: Fast Kernel Context Isolation with Implicit Context Switches. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. https://doi.org/10.1109/SP46215.2023.00086

[59] Istio Project. 2023. WebAssembly in the Istio Proxy (Envoy). https://istio.io/latest/docs/concepts/wasm/. Accessed: 2022-08-10.

[60] Weizhong Qiang, Yong Cao, Weiqi Dai, Deqing Zou, Hai Jin, and Benxi Liu. 2017. Libsec: A Hardware Virtualization-Based Isolation for Shared Library. In *Proceedings of the International Conference on High Performance Computing and Communications (HPCC)*. https://doi.org/10.1109/HPCC-SmartCity-DSS.2017.5

[61] RedPanda. 2021. Redpanda Wasm engine architecture. https://redpanda.com/blog/wasm-architecture.

[62] Jerome H. Saltzer. 1974. Protection and the Control of Information Sharing in Multics. *Commun. ACM* 17, 7 (jul 1974), 388–402. https://doi.org/10.1145/361011.361067

[63] Jerome H Saltzer and Michael D Schroeder. 1975. The protection of information in computer systems. *Proc. IEEE* 63, 9 (1975).

[64] Dylan Schiemann. 2020. Zoom on Web: WebAssembly SIMD, WebTransport, and WebCodecs. https://www.infoq.com/news/2020/08/zoom-web-chrome-apis/.

[65] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. 2022. Jenny: Securing Syscalls for PKU-based Memory Isolation Systems. In *Proceedings of the USENIX Security Symposium (USENIX Security)*.

[66] 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)*.

[67] Martin Schwarzl, Pietro Borrello, Andreas Kogler, Kenton Varda, Thomas Schuster, Michael Schwarz, and Daniel Gruss. 2022. Robust and Scalable Process Isolation Against Spectre in the Cloud. In *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*.

[68] Mark Seaborn. 2013. Sandboxing Libraries in Chrome using SFI: zlib Proof-of-Concept. https://docs.google.com/presentation/d/1RD3bxsBfTZOIfrlq7HzGMsygPHgb61A1eTdelIYOurs/.

[69] Simon Shillaker and Peter Pietzuch. 2020. Faasm: Lightweight isolation for efficient stateful serverless computing. In *Proceedings of the USENIX Annual Technical Conference (ATC)*.

[70] Mincheol Sung, Pierre Olivier, Stefan Lankes, and Binoy Ravindran. 2020. Intra-Unikernel Isolation with Intel Memory Protection Keys. In *Proceedings of the ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*. https://doi.org/10.1145/3381052.3381326

[71] Gang Tan. 2017. Principles and Implementation Techniques of Software-Based Fault Isolation. *Foundations and Trends in Privacy and Security* 1, 3 (2017). https://doi.org/10.1561/3300000013

[72] The Cheri team 2022. CHERI Clang/LLVM and LLD. https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheri-llvm.html.

[73] The Cheri team 2022. CheriBSD. https://github.com/CTSRD-CHERI/cheribsd.

[74] The IBM i architecture 2022. Getting started with IBM i. https://developer.ibm.com/articles/i-newtoibmi/.

[75] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *Proceedings of the USENIX Security Symposium (USENIX Security)*.

[76] Kenton Varda. 2018. WebAssembly on Cloudflare Workers. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/.

[77] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. 2022. You shall not (by) pass! practical, secure, and fast PKU-based sandboxing. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. https://doi.org/10.1145/3492321.3519560

[78] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. 1993. Efficient Software-Based Fault Isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles* (Asheville, North Carolina, USA) *(SOSP '93)*. Association for Computing Machinery, New York, NY, USA, 203–216. https://doi.org/10.1145/168619.168635

[79] Evan Wallace. 2017. WebAssembly cut Figma's load time by 3x. https://www.figma.com/blog/webassembly-cut-figmas-load-time-by-3x/.

[80] Xiaoguang Wang, SengMing Yeoh, Pierre Olivier, and Binoy Ravindran. 2020. Secure and Efficient In-Process Monitor (and Library) Protection with Intel MPK. In *Proceedings of the European Workshop on Systems Security (EuroSec)*. https://doi.org/10.1145/3380786.3391398

[81] Nicholas C Wanninger, Joshua J Bowden, Kirtankumar Shetty, Ayush Garg, and Kyle C Hale. 2022. Isolating functions at the hardware limit with virtines. In *Proceedings of the European Conference on Computer Systems (EuroSys)*. https://doi.org/10.1145/3492321.3519553

[82] Ofir Weisse, Ian Neal, Kevin Loughlin, Thomas F. Wenisch, and Baris Kasikci. 2019. NDA: Preventing Speculative Execution Attacks at Their Source. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1145/3352460.3358306

[83] Jonathan Woodruff, Alexandre Joannou, Hongyan Xia, Anthony Fox, Robert M. Norton, David Chisnall, Brooks Davis, Khilan Gudka, Nathaniel W. Filardo, A. Theodore Markettos, Michael Roe, Peter G. Neumann, Robert N. M. Watson, and Simon W. Moore. 2019. CHERI Concentrate: Practical Compressed Capabilities. *IEEE Trans. Comput.* 68, 10 (2019), 1455–1469. https://doi.org/10.1109/TC.2019.2914037

[84] 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

[85] Mengjia Yan, Jiho Choi, Dimitrios Skarlatos, Adam Morrison, Christopher Fletcher, and Josep Torrellas. 2018. Invisispec: Making speculative execution invisible in the cache hierarchy. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. https://doi.org/10.1109/MICRO.2018.00042

[86] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*. https://doi.org/10.1109/SP.2009.25

[87] Jiyong Yu, Mengjia Yan, Artem Khyzha, Adam Morrison, Josep Torrellas, and Christopher W. Fletcher. 2020. Speculative Taint Tracking (STT): A Comprehensive Protection for Speculatively Accessed Data. *IEEE Micro* 40, 3 (2020), 81–90. https://doi.org/10.1109/MM.2020.2985359

[88] Zakai. Alon 2020. WasmBoxC: Simple, Easy, and Fast VM-less Sandboxing. https://kripken.github.io/blog/wasm/2020/07/27/wasmboxc.html.

[89] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. 2014. ARMlock: Hardware-based fault isolation for ARM. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS)*. https://doi.org/10.1145/2660267.2660344

Received 2022-10-20; accepted 2023-01-19