THEME ARTICLE: TOP PICKS FROM THE 2023 COMPUTER ARCHITECTURE CONFERENCES

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

Shravan Narayan, *UT Austin, Austin, TX, 78712, USA*
Tal Garfinkel, *UC San Diego, San Diego, CA, 92093, USA*
Mohammadkazem Taram, *Purdue University, West Lafayette, IN, 47907, USA*
Joey Rudek, *UC San Diego, San Diego, CA, 92093, USA*
Daniel Moghimi, *Google, Mountain View, CA, 94043, USA*
Evan Johnson, *UC San Diego, San Diego, CA, 92093, USA*
Chris Fallin, *Fastly, San Francisco, CA, 94107, USA*
Anjo Vahldiek-Oberwagner, *Intel Labs, Berlin, Berlin, 14167, Germany*
Michael LeMay, *Intel Labs, Hillsboro, OR, 97124, USA*
Ravi Sahita, *Rivos, Santa Clara, CA, 95054, USA*
Dean Tullsen, *UC San Diego, San Diego, CA, 92093, USA*
Deian Stefan, *UC San Diego, San Diego, CA, 92093, USA*

Abstract—Hardware-assisted Fault Isolation (HFI) is a minimal extension to current processors that supports secure, flexible, and efficient in-process isolation. HFI addresses the limitations of software-based fault isolation (SFI) systems including: runtime overheads, limited scalability, vulnerability to Spectre attacks, and limited compatibility with existing code and binaries. HFI can be seamlessly integrated into exisiting SFI systems (e.g. WebAssembly), or directly sandbox unmodified native binaries. To ease adoption, HFI proposes incremental changes to existing high-performance processors.

Introduction

Lightweight in-process isolation can change how we organize software systems for improved security, flexibility, and performance. In recent years, in-process isolation has become ubiquitous thanks to the emergence of WebAssembly (Wasm). In the browser, Wasm powers applications used by billions of people daily (e.g. Zoom, Figma, Photoshop). Beyond the browser, Wasm is enabling isolation in novel use cases where existing hardware-based protection can't—from hyper-consolidated FaaS platforms and high-performance data planes, to data streaming platforms[1].

Wasm makes these novel use cases possible by enforcing isolation in software—using Software-based Fault Isolation (SFI). SFI enforces isolation through a combination of compiler instrumentation and virtual memory tricks. Thus, it allows Wasm to avoid the high context-switch overheads, slow cold-starts, scaling limits, etc. of primitives such as processes, containers and VMs that rely on existing hardware protection mechanisms (e.g., page tables[2,3]).

For example, Wasm context switches are very fast—in the low 10s of cycles[4], roughly the same as a function call—and orders of magnitude cheaper than a hardware context switch, let alone IPC. Fast context-switches enable tight integration with high performance

XXXX-XXX © 2024 IEEE
Digital Object Identifier 10.1109/XXX.0000.0000000

applications, allowing Wasm to provide safe extensibility in micro-service dataplanes (Istio), realtime databases (SingleStore) data streaming platforms (RedPanda), and SaaS applications (Shopify); it also enables sandboxing to be retrofitted into existing applications without requiring costly re-architecting, for example, to sandbox potentially vulnerable third-party C libraries in Firefox[5].

Similarly, context creation is also very fast — production FaaS systems can spin up a new Wasm instance in 5 μs[1], instead of the tens to hundreds of milliseconds it takes to spin up a container or VM. Along with low context-switch overheads, this has enabled a new class of high-concurrency, low-latency edge computing platforms from Fastly, Cloudflare, Akamai, etc.

While these unique capabilities have opened the door to many new use cases — Wasm's utility is also hampered by limitations that are intrinsic to SFI today including: performance overheads, scaling limitations, limited compatibility with existing code and binaries, and Spectre Safety.

These limitations are not inherent to in-process isolation, rather, they are a byproduct of SFI's attempts to bridge, in software, the gap between past models of hardware protection and the current needs of software systems. To overcome these limitations, we developed *hardware-assisted fault isolation* (HFI)–a minimal set of non-intrusive architecture extensions that bring first-class support for in-process isolation to modern processors.

HFI offers primitives that systematically eliminate typical software (and hardware) isolation 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).

HFI provides first-class assistance for Wasm and similar systems by offering secure, scalable, and efficient hardware primitives that can be used as a drop-in replacement for SFI—it also provides first-class support for backwards compatible in-process isolation, allowing it to sandbox existing native binaries and dynamically generated code. HFI achieves this with minimal additional hardware and minor changes to the control and data paths of existing processors, making it easy to adopt.

Several key design choices enable these unique properties:

1) 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.

2) HFI does not rely on the MMU for in-process isolation — instead, sandboxing is enforced via a new mechanism called *regions*; regions enable coarse-grain isolation (e.g., heaps) and fine-grain sharing (e.g., objects) within a processes' address space.

3) 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[2,6].

SFI and its Limitations

SFI is the dominant (and in terms of practical deployment, the only) technology for fine grain in-process isolation today. This is, in part, because isolation that relies on modern page-based protection (e.g. processes, VMs) are poorly suited to fine grain isolation for a variety of well known reasons[3] including: expensive context switches due to protection ring transitions, heavy weight context saves and restores[4], increased TLB flushes and contention as concurrency scales, etc.

SFI[7] — and by extension, Wasm — avoids these costs by using compiler-added instrumentation, rather than hardware protection—to enforce isolation by interposing on all memory access. Conceptually, SFI is simple: memory is viewed as a set of contiguous memory regions with a base (starting address) and a size—a compiler adds the base address of a sandbox to the operand of any memory operation, e.g., load, then checks that the result is in-bounds, rather like a poor man's version of segmentation.

Naively, we could imagine implementing this by adding explicit bounds checks to each memory load/store and instruction fetch, however, this can easily slow down code by a factor of 2$\times$[7]. Instead, Wasm and other modern SFI systems (e.g., Native Client) rely on a faster technique which relies on the MMU to enforce bounds implicitly using a system of large address spaces and guard regions.

Wasm runtimes accomplish this by allocating a 4 GiB address space (called a *linear memory* in Wasm), followed by a 4 GiB guard region (unmapped address space) for each sandbox. When accessing linear memory, each Wasm load/store instructions instruction takes two 32-bit unsigned operands. A Wasm compiler will generate code to add these operands, resulting in a 33 bit address—and add the result to a 64bit *base address*—the starting address of a linear memory. By construction, any 33 bit unsigned offset plus a base

address will be within 8GB of the base; thus, access beyond the first 32-bit (4GB) address space will trap. To isolate control flow, Wasm also relies on software control flow integrity[8].

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

32-bit address spaces. Guard region based SFI only works for 32-bit address spaces on 64-bit architectures — supporting larger Wasm sandboxes, or smaller processors, requires falling back to explicit bounds checks with the high overheads (up to 2x) this implies.

Performance overheads. Even with these tricks, Wasm can still easily impose performance overheads of 40% — sometimes less, and sometimes a lot more[5]. Some costs are fundamental to SFI, such as restrictions on the formats of memory instructions and added register pressure[2]. Some are specific to Wasm, such as the cost of software CFI, and limited access to SIMD instructions.

Spectre. Wasm cannot protect itself against Spectre attacks without performance penalties — to wit — software-based mitigations add an additional 62% to 100% of overhead[9].

Compatibility. A compiler must explicitly target Wasm to use it. Thus, assembly language, platform specific compiler intrinsics, dynamically generated code, and existing binaries (e.g. precompiled libraries) are not supported.

Scaling (Virtual memory consumption). As previously noted, every Wasm instance consumes 8 GiB of virtual address space, even if it only uses only a few 100 MB or less, as most serverless edge workloads do today.

This limits scaling as virtual address space is finite — typical x86-64 CPUs provide 2^{47} (128 TiB) worth of user-accessible virtual address space[1]. Thus, at 8 GiB (2^{33}) per-instance we can run at most 16K (2^{14}) instances concurrently per-process. High performance edge computing platforms are already running up against this limit today. These systems spin up a new instance in μ-seconds for every incoming network request, and requests often block for I/O; thus massive concurrency is the norm.

At present, their only recourse is to spin up more processes, and load balance requests between them. Unfortunately, this leads to load imbalances and expensive context switch overheads as processes contend for physical cores. Also, 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)[4].

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.

Design Overview

Hardware-assisted Fault Isolation (HFI) is a minimal extension to current processors for secure, flexible, and efficient in-process isolation. For this, HFI introduces architecture primitives to create one or more in-process sandboxes. These can be either *hybrid* sandboxes, that integrate HFI's primitives into existing SFI compilers like Wasm, for increased performance, scalability, security, etc. or *native* sandboxes, that can confine arbitrary untrusted binaries.

A sandboxing runtime enables HFI with the `hfi_enter` enter instruction, resulting in sandboxing constraints (memory and control isolation) being enforced until the `hfi_exit` instruction is invoked, at which time HFI returns control back to the runtime. The runtime is responsible for saving and restoring the sandbox's context (e.g. general purpose registers), and can multiplex many sandboxes across cores, scheduling them as it sees fit.

A sandbox's semantics are dictated by a set of per-core HFI registers consisting of: (a) region registers that grant access to memory, (b) a register with the sandbox exit handler—where system calls and sandbox exits are redirected—supporting full control of privileged instructions and control flow, and (c) a register with sandbox option flags, e.g. whether the sandbox is a hybrid or native sandbox.

Efficient Memory Access Control with Regions. HFI's memory and control isolation is configured using a finite set of regions. Regions in HFI are *base* and *bound* pairs that identifies a range of memory that can be accessed (e.g. stack, heap, code), and permissions (read, write, execute) that apply to that range.

Notably, a central aspect of HFI's design is using region specialization to minimize the hardware required to bounds check regions, as these checks are located

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

in the heart of the processor's control and datapath, where every additional gate matters.

Concretely, an approach that uses two 64-bit comparators (per region) would be the obvious design choice for ensuring memory accesses are restricted to the configured regions; however, these are large circuits that would add unacceptable delay and power consumption to a processors critical path. Instead, HFI offers multiple region types, each of which reduces hardware complexity by being specialized to a particular task. We will describe the different region types, then discuss how they are used.

HFI uses **implicit regions** to 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, else HFI will trap.

Implicit regions are essential for situations when every memory operation in an application must be checked. In exchange for this power, they assume some constraints on a region's size and alignment, in particular, implicit regions require a region's address to occupy an aligned location and have a size which is a power of 2.

This restriction typically requires developers to only make slight modifications to how memory allocation occurs; however, it allows hardware to implement the region checks as simple, fast masking operations. Again, for hardware simplicity, implicit regions are further specialized into code and data regions. In total, HFI provides 4 implicit data regions, and 2 implicit code regions for a sandbox.

HFI also provides 4 **explicit data regions**, that trade the generality of implicit regions for precision. Specifically, all loads and stores to explicit regions are region-relative, i.e., name the region they apply to in using the new `hmov[1-4]` instruction, a variant of the x86-64 `mov` instruction, with the region number encoded in the instruction.

In exchange for this constraint, the size and alignment constraints of implicit regions are relaxed, allowing more precise control over memory layout. There are two types of explicit regions, small—that support byte granular sizes alignment, and are limited to a max of 4GB, and large—which are 64K-aligned and can be any multiple of 64K in size. This added specialization again supports simpler hardware; explicit regions can be supported with just a single 32-bit comparator. In total, HFI provides 4 explicit data regions for a sandbox.

Using Regions. By default, a sandbox cannot access any memory—region registers must be set before sandbox entry (`hfi_enter`), to grant memory access to a sandbox using a handful of new instructions—`hfi_set_region`, `hfi_get_region`, `hfi_clear_region`.

Using the above instructions to configure HFI's explicit and implicit regions meets a diverse set of needs. HFI's implicit regions are ideal for sandboxing unmodified native binaries, where large regions of memory (e.g. the stack, heap, code) are not as particular about their size or alignment. While the added precision of explicit regions makes them well suited to supporting heaps for SFI systems like Wasm — where heaps grow in fixed size increments, and all operations are already explicitly relative to a region of memory — as well as for fine-grain, zero-copy sharing of objects between two different sandboxes.

System call interposition is another important feature of HFI. Architecture support makes this every efficient (`syscall` instructions are optionally converted to jumps to a specified location in the processor's decode stage), simple, and accessible at user level. With this, HFI can sandbox unmodified native binaries, while ensuring sandboxing is not bypassed or disabled[10]. System call interposition is used by setting a flag in the HFI option register prior to sandbox entry, as well as providing an exit handler — the location where system calls are redirected to.

HFI offers a variety of other features to enable secure and efficient sandboxing, for example, to mitigate certain classes of Spectre attacks. Please consult our full paper for more details[11].

μArchitecture design

Five overarching goals guide and constrain the design of HFI's μ-architecture:

1) **Fast.** Minimizing overhead is a central goal of HFI. Thus, HFI should add any new circuits such that they are run concurrently with existing operations such as TLB lookups.

2) **Secure.** HFI must be robust to Spectre, 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).

3) **Scalable.** HFI must not store any state on-chip that scales with the number of sandboxes. This could restrict the total number of concurrent sandboxes or require expensive state spills on overflow, resulting in performance that scales down as concurrency increases. It

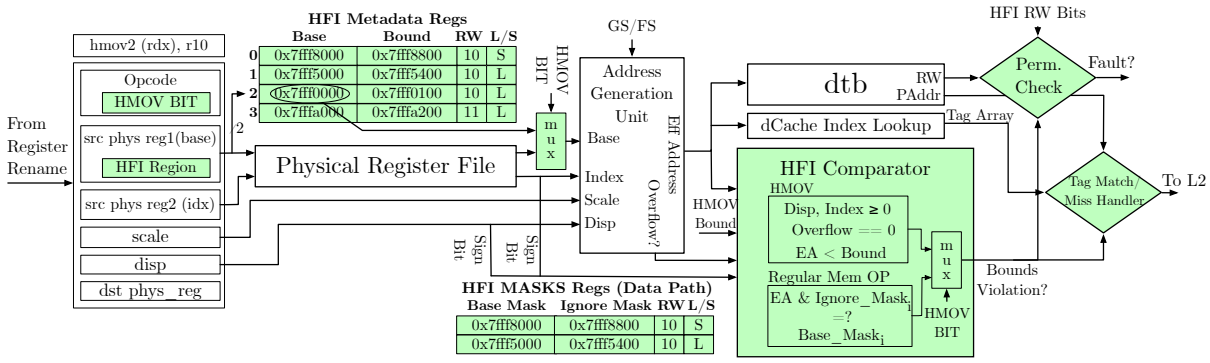

Figure 1. HFI Impact on the x86 data pipeline. We see the x86 data pipeline with added HFI components in Green. HFI adds no additional overhead to the data path — no new pipeline stages are added — and all new operations take place in parallel with the dtb (dTLB) lookup or instruction decode stages.

must also minimize context switch overheads by limiting on-chip state for the current sandbox.

4) **Minimal.** HFI enforces region bounds checks in the neighborhood of critical pipeline structures such as the address generation unit, and the TLB; thus it must not add new large structures (even if the new structures are not on the critical timing path, it can push components physically further apart, which in turn affects timings and the CPU frequency). Thus, HFI must avoids components like 64-bit comparators that would otherwise be natural design choices.

5) **Tax-free.** HFI should not impose slowdowns on code that doesn't use it, given that most code today is not sandboxed. Thus, HFI must not add extra pipeline stages, nor timing delays to existing stages on the critical timing path of the processor.

Given these constraints, we now briefly outline the implementation of HFI's essential components — the regions (implicit data and code regions, and explicit data regions), sandbox setup, and system call interposition.

Implicit Regions. HFI allocates two registers for each implicit region, for the `lsb_mask` and `base_prefix`. If this is an implicit data region, checks are applied to all load and store operations on the data path if this is an implicit code region, checks are applied to the program counter.

Concretely, these checks work by *prefix matching* — 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, checks for the implicit data region occur in parallel with the dtb and cache index lookup; checks for the implicit code region are applied in parallel with the decode stage.

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, similar to a memory access to unmapped memory.

Explicit Data Regions. Explicit data regions are regions accessed with the new `hmov` (`hmov0`, `hmov1`, `hmov2`, `hmov3`) instructions, that confines data accesses to the specified region. `hmov` operates like a standard x86 `mov`, including complex addressing modes where *scale, index, base* and *displacement* operands are added/combined to form the effective address. In our paper[11], we proposed implementing `hmov` by adding three additional steps that: (1) choose an HFI region (encoded in the instruction), (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 (Figure 1). However, step (2) limits the flexibility of the `mov` instruction. In subsequent research, we realized we can recover this flexibility by using the segment operand (which is rarely used) rather than the base operand for addressing our region.

Explicit bounds checks are implemented with a single 32-bit comparator and a few cheap bit-level checks. 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 single-bit checks).

This approach works well as long as applications create regions that follows some simple constraints: for large regions, HFI works as long as the base and bounds are aligned to 64K (2^{16}) on the typical x86 CPUs that support using 48-bits out of the 64-bit virtual address space[2]; for small regions (upto 4 GiB), HFI only checks the bottom 32-bits of the effective address against the base, and can thus support arbitrary bounds as long as the region does not span across an address that is a multiple of 4 GiB.

Sandbox setup. HFI adds region configuration instructions previously discussed (e.g. `hfi_set_region`) that manipulate internal region registers, as well as `hfi_enter` and `hfi_exit` to enable and disable sandboxing. The `hfi_enter` instruction saves its parameters — the exit handler and flags to internal configuration registers, enables HFI mode. On exit, HFI disables the sandboxing, and 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.

System call interposition. HFI can be configured to redirect syscalls executed by sandboxed code. To implement this, HFI modifies the decode stage of the `syscall` instruction and its variations to perform a microcode check and redirect control flow to the HFI exit handler if this is the case. The `syscall` instruction is otherwise unmodified. The single cycle penalty this imposes on syscall instructions is amoritized by the typical costs of standard system calls.

Evaluation

We implemented the HFI in the Gem5 simulator for detailed performance analysis. We also used compiler-based emulation (that emits instructions such as `cpuid`, `lfence` etc. to insert appropriate slowdowns) to efficiently evaluate longer running workloads and validated that our emulator has similar performance characteristics as our Gem5 simulator. Please consult our full paper for more details[11].

Performance Evaluation

We evaluated the performance of HFI in four use cases: long-running applications (SPEC), library sandboxing in a browser, a JIT-based FaaS, and native sandboxing in a server workload (NGINX).

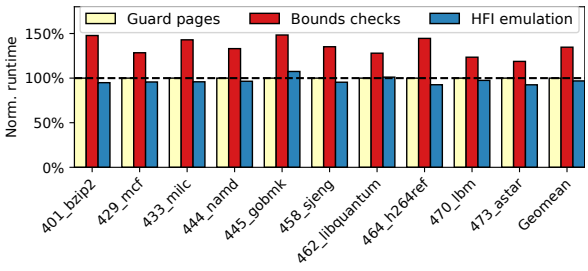

Figure 2. SPEC INT 2006 results normalized against guard pages.

SPEC 2006 Benchmark Suite. We evaluate HFI on the subset of SPEC06 that is compatible with Wasm. Figure 2 shows the performance Wasm with explicit bounds checks, guard pages, and HFI—relative to the cost of guard pages.

These long-running applications do not test HFI's fast transitions, but do show its low cost in steady state. 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. However, HFI is the only scheme of these three that also offers Spectre protection, and mitigating Spectre without HFI, incurs a 62% to 100% penalty[9].

Wasm Sandboxing in Firefox. To understand the end-to-end performance impact of HFI, we evaluate the performance of HFI in sandboxing font rendering and sandboxed image rendering in Firefox. Notably, both these benchmarks stress HFI's transitions as the code rapidly jumps from the sandboxed code to Firefox code — one per glyph/letter for font rendering, and once per row for image rendering. The font rendering benchmark reflows the text on a page ten times via the sandboxed `libgraphite`, and takes 1823 ms when using Wasm with guard pages; 2022 ms when using Wasm with bounds-checking; and 1677 ms when Wasm powered by HFI — a 17% and 8% speedup respectively. For Wasm-sandboxed `libjpeg`, 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 3 shows the median decode times for each configuration

[2]On Intel server CPUs that support 52/57-bit address spaces, a larger 36/41-bit comparator would be necessary

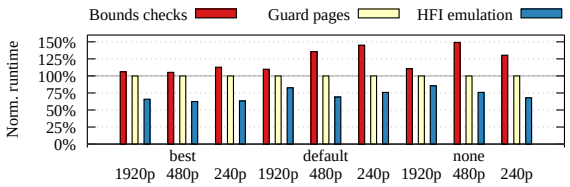

Figure 3. Firefox image rendering. HFI offers a significant speedup for image rendering — the biggest increase for larger images that amortize the cost of `hfi_enter`. More compressed images — that are more compute intensive — also see greater benefits, as a result of decreased register pressure.

out of 1000 runs. As expected, HFI offers the fastest sandboxing compared to the typical software-based enforcement of Wasm.

HFI and Scalability. As HFI eliminates the need for guard regions, it can support far more concurrent instances in a process' address space, a highly desirable trait in high scale serverless settings. To exercise this, we modified Wasmtime, a Wasm runtime popular in serverside settings, to use HFI instead of guard regions, and spun up as many 1 GiB sandboxes as it would support. As expected, it could create up to 256,000 1 GiB sandboxes in a single process, making full use of the process' address space [3].

HFI for Sandboxing Native Binaries. We used HFI to sandbox a native binary and compared it ERIM[6], a state-of-the-art system for sandboxing. ERIM uses Intel Memory Protection Keys (MPK) for sandboxing memory accesses, and Linux's seccomp-BPF to filter system calls.

We first compare the overhead of ERIM's seccomp-BPF system call filter to HFI's microarchitectural support for the same. For this, we ran a custom syscall benchmark that opens a file, reads it, and closes it 100,000 times. We found that using seccomp-BPF version imposes an overhead of 2.1%, over HFI.

Next, we modify the NGINX webserver to estimate the performance of sandboxing crypto functions and session keys in OpenSSL, similar to ERIM[6]. We used this to measure the costs of integrating HFI in existing applications versus the benefits (e.g., blocking attacks like Heartbleed and Spectre).

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_region`), and traps. We compare the throughput of the NGINX web server delivering content when protecting session keys with HFI and MPK respectively. HFI's overhead ranges from 2.9% to 6.1%. compared to MPK's range from 1.9% to 5.3% — a slight increase due to the cost of HFI's metadata manipulation as well as serialization on `hfi_enter` and `hfi_exit` for Spectre protection.

Spectre Evaluation

We use proof-of-concept attacks from the TransientFail (the in-place Spectre-PHT attack) and Google SafeSide (the in-place Spectre-BTB attack) test suites to ensure HFI is resistance to Spectre attacks. HFI prevents both these attacks ensuring sandboxed code cannot speculatively access secret data (stored in a global variable in the host application for this examples).

We compared HFI's Spectre protection overheads, to the performance of Swivel[9], the fastest known compiler-based approach to mitigating Spectre in Wasm. We evaluated this by measuring costs of Spectre protection on several common FaaS Wasm workloads running in the Rocket webserver. We compiled our Wasm workloads with a Lucet Wasm compiler without Spectre protections, Lucet+Swivel-SFI protections, and with Lucet+HFI using native sandbox. Finally, we also record the binary sizes of all three builds.

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.

Path to Adoption

HFI offers a unique solution for hardware-assisted isolation that balances novel acceleration capabilities with practical adoption.

Previous approaches to sandboxing that re-purpose existing hardware[6] such as Memory Protection Keys, consistently fall short with respect to scaling, performance, security[10], etc. On the other hand, ambitious novel architectures such as CHERI impose high costs in terms of hardware complexity, and modifying existing software stacks, that pose significant barriers to adoption. HFI illustrates a middle path between these two extremes, offering dedicated hardware acceleration for isolation that is timely, relevant, and easy to adopt.

[3]Wasmtime normally supports up to 21,504 sandboxes. It is able to slightly exceed the 16K limit through a sophisticated combination of guard regions and bounds checks.

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

In recent years, SFI, in the form of Wasm, has seen broad adoption on the web; in edge computing platforms; for sandboxing of buggy C libraries[5]; and providing safe extensibility in numerous applications, and by web3 platforms to run decentralized applications. Beyond its use in Wasm, SFI is also used by Chrome to isolate their V8 JavaScript Engine; and by the Linux kernel (eBPF) to provide extensibility, performance analysis, and security monitoring.

This widespread use of SFI allows HFI to avoid the classic chicken-and-egg problem faced by new hardware security features — there is already a huge base of software using SFI that can (with minor changes) immediately benefit from the added security, performance, scalability, etc. that HFI offers.

Further, HFI was refined through numerous iterations of feedback from architects at Intel to determine what changes are practical in existing high-performance processors. As a result, HFI offers precisely what is needed to support in-process isolation with a minimal hardware bill of materials.

Additionally, HFI requires very little support from the OS (on the order of tens of lines of code), again informed by the reality that getting OS kernel support is a significant barrier to rolling out new hardware features.

Our work with HFI is just beginning, engineers who develop Chrome and Firefox are excited about the benefits that HFI can bring to their hardening efforts[12]; engineers at edge computing companies are excited about its potential to improve scalability, security, and performance on their platforms. And processor architects are excited about enabling these benefits with minimal hardware costs.

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 the original published 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, Cisco, and Mozilla; and by DARPA HARDEN under NIWC contract N66001-23-9-4004. And finally, thanks to our families, without whose support this work would not be possible.

References

1. L. Clark, "Wasmtime reaches 1.0: Fast, safe and production ready!." https://bytecodealliance.org/articles/wasmtime-1-0-fast-safe-and-production-ready, Sept. 2022.
2. G. Tan, "Principles and implementation techniques of software-based fault isolation," *Foundations and Trends in Privacy and Security*, vol. 1, no. 3, 2017.
3. R. M. Norton, "Hardware support for compartmentalisation," tech. rep., University of Cambridge, Computer Laboratory, 2016.
4. M. Kolosick, S. Narayan, C. Watt, M. LeMay, D. Garg, R. Jhala, and D. Stefan, "Isolation without taxation: Near zero cost transitions for SFI," in *POPL*, 2022.
5. S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan, "Retrofitting fine grain isolation in the Firefox renderer," in *USENIX Security*, 2020.
6. A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in *USENIX Security*, 2019.
7. R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *SOSP*, 1993.
8. A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," in *PLDI*, 2017.
9. S. Narayan, C. Disselkoen, D. Moghimi, S. Cauligi, E. Johnson, Z. Gang, A. Vahldiek-Oberwagner, R. Sahita, H. Shacham, D. Tullsen, and D. Stefan, "Swivel: Hardening WebAssembly against Spectre," in *USENIX Security*, 2021.
10. R. J. Connor, T. McDaniel, J. M. Smith, and M. Schuchard, "PKU pitfalls: Attacks on PKU-based memory isolation systems," in *USENIX Security*, 2020.
11. S. Narayan, T. Garfinkel, M. Taram, J. Rudek,

D. Moghimi, E. Johnson, C. Fallin, A. Vahldiek-Oberwagner, M. LeMay, R. Sahita, D. Tullsen, and D. Stefan, "Going beyond the limits of SFI: Flexible and secure hardware-assisted in-process isolation with HFI," ASPLOS 2023.

12. S. Groß (saelo), "V8 sandbox - hardware support." https://docs.google.com/document/d/12MsaG6BYRB-jQWNkZiuM3bY8X2B2cAsCMLLdgErvK4c/, 2024.

Shravan Narayan is an Assistant Professor at University of Texas at Austin, Austin, TX, 78712, USA. His research interests include building secure systems, program verification, and hardware-based security. He received his doctorate degree from the University of California, San Diego. Contact him at shr@cs.utexas.edu.

Tal Garfinkel Tal Garfinkel is a Research Scientist at UC San Diego. He is interested in hardware/software systems from programming languages to computer architecture, and how these systems can be improved for greater security, reliability, and performance. He received his doctorate degree from Stanford University. Contact him at talg@cs.ucsd.edu.

Mohammadkazem Taram is an Assistant Professor at Purdue University. His research interests are in computer architecture and computer security. In particular, he is interested in microarchitectural attacks, high-performance mitigations, and architecture support for security and privacy. Contact him at kazem@purdue.edu.

Joey Rudek is a PhD student at University of California, San Diego. His research focuses on secure hardware design and the security implications of novel hardware optimizations. Contact him at jrudek@ucsd.edu.

Daniel Moghimi is a Senior Research Scientist at Google. His research focusses on computer and hardware security, spanning various topics such as microarchitectural vulnerabilities, side-channel cryptanalysis, and security architecture. He received his doctorate degree from Worcester Polytechnic Institute, and was a postdoctoral scholar at the University of California, San Diego. Contact him at danielmm@google.com.

Evan Johnson is a PhD student at University of California, San Diego. His research focuses on software security and software correctness. In particular, he works on securing and verifying software sandboxing systems. Contact him at e5johnso@ucsd.edu.

Chris Fallin is a Principal Software Engineer at Fastly on the WebAssembly team. He focuses on compilers, runtimes, and system software with an emphasis on correctness. He served as the tech lead for the Cranelift JIT compiler in the Wasmtime virtual machine, and is leading formal verification and other testing efforts and novel dynamic-language implementation techniques on the platform. He received his doctorate degree from Carnegie Mellon University, where he focused on static and dynamic analysis. Contact him at chris@cfallin.org.

Anjo Vahldiek-Oberwagner is a research scientist with Intel Labs in the Datacenter Security Group. He focuses on analyzing and building secure software and hardware computing systems. In particular, he develops PoCs demonstrating the security, performance and usability of in-process isolation in datacenter workloads. He received his doctorate degree from the Max Planck Institute for Software Systems and Saarland University. Contact him at anjo.lucas.vahldiek-oberwagner@intel.com.

Michael LeMay is a senior staff research scientist in Intel Labs. His research interests include memory management architectures for security. LeMay received his doctorate degree from the University of Illinois at Urbana-Champaign. Contact him at michael.lemay@intel.com.

Ravi Sahita is a Principal Member of Technical Staff at Rivos Inc where he leads the platform security architecture and ISAs, as well as platform and software initiatives. He serves as the vice-chair of the Security horizontal committee at RISC-V International, and leads the Confidential Computing SIG and Task Groups. In past work as Sr. Principal Engineer at Intel, he led the definition of confidential computing on Intel architecture (Intel® Trust Domain Extensions), exploit prevention ISA (control flow integrity and virtualization-based security). He has authored technical standards in RISC-V, IETF and TCG, has been granted 220 patents, and is the recipient of two Intel Achievement Awards. Contact him at ravi@rivosinc.com.

Dean Tullsen is a professor at UC San Diego. His research interests include computer architecture and architectural security. With his various collaborators he has made several contributions to the architecture of modern processors, including simultaneous multithreading and heterogeneous multicore architectures. He is a Fellow of the ACM and a Fellow of the IEEE. Contact him at tullsen@ucsd.edu.

Deian Stefan is an Associate Professor in the University of California, San Diego. He is interested in research that builds principled and practical secure systems using

techniques that span security, programming languages, and systems. His recent work includes secure systems (from Web frameworks, to new browser designs, sandboxing, and runtime systems), language-based security (constant-time programming, memory safety, and information flow control), verification for security, and (static and symbolic) program analysis tools. He received his doctorate degree from Stanford University. Contact him at deian@cs.ucsd.edu.