Hardware-Assisted Fault Isolation: Going Beyond the Limits of Software-Based Sandboxing

Shravan Narayan, *University of Texas at Austin, Austin, TX, 78712, USA*
Tal Garfinkel, *University of California San Diego, San Diego, CA, 92093, USA*
Mohammadkazem Taram, *Purdue University, West Lafayette, IN, 47907, USA*
Joey Rudek, *University of California San Diego, San Diego, CA, 92093, USA*
Daniel Moghimi, *Google, Mountain View, CA, 94043, USA*
Evan Johnson, *University of California 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, *University of California San Diego, San Diego, CA, 92093, USA*
Deian Stefan, *University of California 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 existing 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 existing SFI systems (e.g. WebAssembly), or directly sandbox unmodified native binaries. To ease adoption, HFI relies only on incremental changes to existing high-performance processors.

Introduction

Isolation primitives have a critical impact on how we organize software systems for security, reliability, flexibility, and performance.

Today, familiar isolation primitives such as virtual machines, processes, and containers, all rely on a common set of underlying hardware primitives, i.e., page tables and protection rings. However, in recent years, software developers have been reaching for another technique, Software-based Fault Isolation (SFI)[1,2]—to overcome the limitations of existing primitives.

SFI enforces isolation in-process, i.e., it allows the creation of isolated "sandboxes" in a process (or kernel) address space, through a combination of compiler instrumentation and virtual memory tricks. SFI is attractive as it avoids the limitations of existing hardware based isolation primitives[1,3] such as high context-switch overheads, slow cold-starts, and scaling limitations.

With the wide-spread adoption of WebAssembly (Wasm), SFI has become ubiquitous. In the browser, Wasm powers applications used by billions of people daily (e.g. Zoom, Figma, Photoshop). Beyond the browser, developers are using Wasm to support use cases that existing isolation primitives can't—thanks to the novel capabilities SFI offers.

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

To start, SFI supports context-switches for roughly the same cost as a function call — orders of magnitude cheaper than existing isolation primitives. This enables isolation that is tightly integrated with high performance applications. For example, Wasm is used to provide extensibility in micro-service dataplanes (Istio), real-time databases (SingleStore) data streaming platforms (RedPanda), and SaaS applications (Shopify). This tight integration also enables sandboxing to be retrofitted into existing applications—to contain the impact of memory safety vulnerabilities—without costly re-engineering. For example, Firefox[4] now relies on Wasm to sandbox the third-party C libraries it relies on for audio decoding, XML parsing, spell checking, and many other tasks.

SFI also enables fast context creation. For example, production function-as-a-service (FaaS) systems can spin up a new Wasm instance in 5 μs[5], instead of the tens to hundreds of milliseconds it takes to spin up a container or virtual machine. This has enabled a new class of high-concurrency, low-latency edge computing platforms from Fastly, Cloudflare, Akamai, etc.

While SFI's unique capabilities have opened the door to many new use cases — it also has a variety of limitations including performance overheads, scaling limitations, limited compatibility with existing code and binaries, and Spectre Safety. These are the natural result of the fact that SFI is a hack–an ad-hoc software techniques trying bridge 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.

SFI and its Limitations

SFI is the only widely deployed technology for fine grain in-process isolation today. It's unique capabilities are enabling the growth of in-process isolation in many domains, in particular, in the form of WebAssembly. As discussed, SFI achieves this through an ad-hoc software technique which allow it to skirt some of the key limitations[3] of existing hardware isolation—however, this ad-hoc approach brings limitations of its own.

Conceptually, SFI enforces isolation by interposing on all memory access through compiler instrumentation. To implement this, 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$\times$[2] or more. 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 implement 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, that are added, resulting in a 33-bit address—and the result is added to a 64-bit *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[6].

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/SFI can still easily impose performance over-

heads of 40% — sometimes less, and sometimes a lot more[4].

Spectre. Wasm cannot protect itself against Spectre attacks without performance penalties — to wit — software-based mitigations add an additional 62% to 100% of overhead[7].

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} bytes (128 TiB) worth of user-accessible virtual address space[1]. Thus, at 8 GiB (2^{33} bytes) per-instance we can run at most 16,000 (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. 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$\times$ to 10000$\times$ slower across process boundaries due to use of inter-process communication (IPC).

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 can support quite a lot of serverless instances.

After struggling against these limitations for years in the context of building and deploying production library sandboxing and edge computing systems, we began to explore what would be possible with some additional help from hardware. Our resulting hardware extension (HFI) offers all the benefits of SFI, and more, while eliminating these limitations.

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

HFI Overview

Hardware-assisted Fault Isolation (HFI) an instruction set architecture (ISA) extension for modern processors that offers secure, flexible, and efficient in-process isolation, with minimal additional hardware complexity.

Several key design choices enable these unique properties:

1) 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.

2) HFI is fully available in userspace; thus, there are no overheads from ring transitions or system calls when changing memory restrictions, or entering and leaving a sandbox.

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[1,8].

Functionally, HFI allows the creation of one or more in-process sandboxes. These can be either *hybrid* sandboxes, that integrate HFI's primitives into existing SFI system like Wasm, for increased performance, scalability, security, etc. or *native* sandboxes, that can confine arbitrary untrusted binaries without any additional software support—thus shedding some of the compatibility and complexity baggage that accompanies SFI based technologies such as Wasm that rely on significant modifications to existing language toolchains.

To use HFI, a sandboxing runtime uses 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 including: (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.

These checks are located in the heart of the processor's control and datapath, where every additional gate matters. To optimize this, HFI's utilizes multiple types of *specialized regions*.

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. Next, we describe these different region types and 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 very 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[9]. 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[10].

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[10].

Performance Evaluation

We evaluated the performance and scalability of HFI in three use cases: library sandboxing in a browser,

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

a FaaS workload, and native sandboxing in a server workload (NGINX).

Wasm Sandboxing in Firefox. To understand the end-to-end performance impact of HFI, we evaluate the performance of HFI in sandboxing font rendering in Firefox. This benchmark stresses HFI's transitions as the code rapidly jumps from the sandboxed code to Firefox code — one jump per glyph/letter. The 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.

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 [2].

HFI for Sandboxing Native Binaries. We used HFI to sandbox a native binary and compared it ERIM[8], 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 compare the throughput of the NGINX web server delivering content when protecting session keys with HFI and MPK respectively (similar to ERIM[8]). 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.

[2]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.

Spectre Evaluation

We use proof-of-concept attacks from the TransientFail (the in-place Spectre pattern history table attack) and Google SafeSide (the in-place Spectre branch target buffer attack) test suites to ensure HFI is resistance to Spectre attacks. HFI prevents both these attacks ensuring sandboxed code cannot speculatively access secret data.

We compared HFI's Spectre protection overheads, to the performance of Swivel[7], 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.

HFI across different architectures

Our initial HFI prototype[10] was implemented in the context of the x86-64 architecture (See Figure 1). However, it's general design and key principles such as the region specialization to minimize circuit area, avoiding extra pipeline stages, restricting saved state to only the currently executing sandbox etc., are equally applicable to other ISAs, with some modifications to addresses the particular constraints of other ISAs.

For example, on x86s, we added four new `hmov1`, `hmov2`, `hmov3`, `hmov4` instruction prefixes that can be added to any instruction accessing memory, indicating it should operate on one of the explicit data

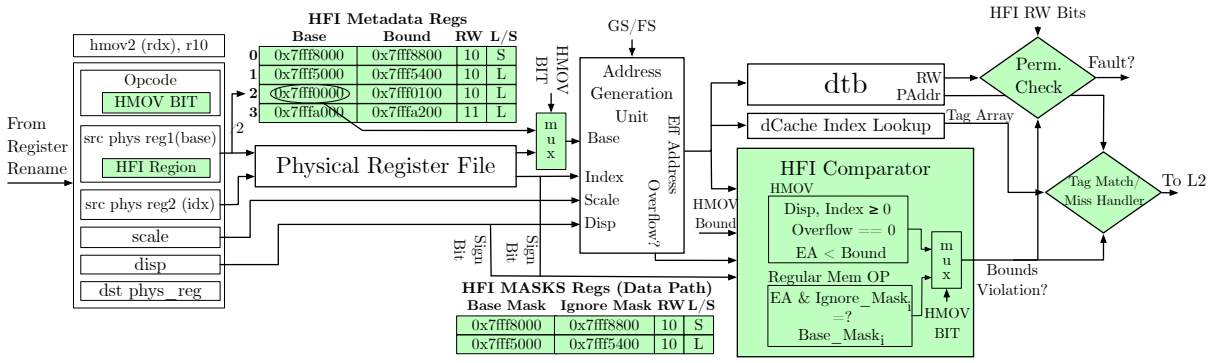

Figure 1. HFI Impact on the x86 data pipeline. The x86 data pipeline with added any HFI components in Green. HFI adds no overhead to the data path — no new pipeline stages are added — and all new operations take place in parallel with the dtb (dTLB) lookup or instruction decode stages.

regions. However, when adapting HFI to RISC-V, we found this approach was incompatible with RISC-V as it does not support instruction prefixes. Rather each RISC-V instruction accessing memory — `load`, `store`, `load_fp`, `store_fp` — would each need four new opcodes to create versions of these instructions accessing explicit regions. This would put unacceptable pressure on RISC-V's already very constrained operation code (opcode) space. Instead, for RISC-V we add a new instruction `h_set_region` to set the active explicit region, and limit `hload`, `hstore` etc. to only modify the active region. While this leads to a less compact instruction encoding in the case where multiple regions are being accessed concurrently, it alleviates this pressure on the op-code space, and also keeps the ISA simpler, keeping with the RISC design philosophy.

Another aspect of our design that differs across ISA's (and processors more generally) is how it is instantiated the microarchitectural level. In particular, HFI's informed by a "living off the land" design philosophy, i.e., we aimed to leverage existing hardware already present in the pipeline as much as possible.

Again, going back to our x86 example, HFI's explicit regions can be implemented by repurposing the circuits for 32-bit segmented memory and 64-bit segment relative addressing. Together these features are implemented using a 32-bit comparator and a 64-bit adder respectively — precisely the circuits needed for HFI's `hmov` instruction — and are both circuits that would be otherwise unused for this instruction. On other architectures, the circuits available for reuse will differ. For example, future RISC-V standards will incorporate support for pointer masking, which could be reused to support implicit regions. While architectures such as the ARM Cortex-M, feature hardware for data watchpoint and trace (DWT)'s range checks, which could be repurposed to implement data region bounds checks.

Applications and Implications

By providing first class support for in-process isolation. HFI opens the door to greater capabilities, and new applications in a variety of domains, here we explore just a few.

Easier Library Sandboxing

Billions of lines of vulnerable C/C++ underlie todays software ecosystem[11]. A cursory inspection of any open source ecosystem of a "safe" language such as Javascript, Java, Rust, Python, etc. reveals thousands of packages that are essentially wrappers around existing C/C++ libraries.

Sandboxing is a potentially powerful tool for mitigating memory safety vulnerabilities in these libraries, as well as other defending against other attack vectors, e.g., supply chain attacks. Unfortunately, sandboxing is rarely utilized, as the high cost of context switches (and consequently IPC) and sandbox creation with process sandboxing constrains the generality of this technique, and often requires significant re-engineering of applications to use it.

While SFI toolchains in such as Wasm have opened the door to retrofitting sandboxing[4], i.e., sandboxing existing libraries in-place without expensive re-engineering—Wasm's many limitations still restrict the usefulness of this technique. In practice, we have seen

that many libraries have hand written assembly code, dynamically generated code, SIMD code that was poorly supported/unsupported, etc.

HFI renders these practical obstacles moot, eliminating many of the performance and engineering overheads that have limited the usefulness of this technique[11].

Secure, Compatible, and Efficient Serverless

Currently serverless infrastructure existing its two radically different forms. On one hand, datacenter serverless architectures like AWS Lambda, Azure Functions, Google Cloudfunctions, etc. rely on heavy weight isolation mechanisms such as VMs and containers to isolate individual instances. While this offers excellent compatibility with existing languages and libraries, and a strong foundation for multi-tenant isolation, startup overheads and latency are often very high (in the 10s to 100s of milliseconds).

On the other, edge computing platforms such Cloudflare workers, and Fastly's Compute@Edge, can offer orders of magnitude improvements in throughput and latency by relying on software based isolation (utilizing Wasm and Javascript)—but in turn sacrifice compatibility. For example, Java and .NET both have mature just-in-time compilers (JITs) and runtimes, that have been optimized and hardened over decades, which cannot be used in these environments.

HFI offers the potential to eliminate this disconnect. Opening the door to future serverless platforms that could offer edge level scaling, efficiency, and responsiveness, while retaining the security and compatibility of existing datacenter serverless platforms.

Secure and Efficient Browser JIT Hardening

One of the hard challenges in browser security today are memory safety bugs in JIT's. Modern Javascript JIT's are marvel's of engineering, offering astonishingly good performance for a dynamic language like Javascript. Unfortunately, the combination of complex optimizations required to achieve this, and the complexity of JavaScript itself, result in a very large attack surface.

Further, some of the best techniques for mitigating these bugs in most applications are not applicable in a JIT context. For example, as bugs are in code optimization and generation, safe languages such as Rust are often of little benefit, since the bugs are in machine code generated by the JIT. Normal Write-xor-execute (W$\oplus$X) restrictions that prevent code injection in normal applications also break in a JIT environment. JIT compilers often have to modify generated executable code; requiring JIT compilers make expensive system-calls to change permissions of memory prior to such changes imposes both a direct slowdown of the JIT compiler, and a system-wide slowdown due to frequent modifications of the translation look-aside buffer (TLB).

One compelling approach to limiting the impact of these bugs is sandboxing JIT's code, for example, Chrome's Ubercage[12]. Unfortunately, such efforts currently have to trade-off security for performance; for example, Ubercage it does not incorporate software protections for control flow hijacking, let alone Spectre, offering multiple potential ways to bypass it.

HFI can advance the state-of-the-art in JIT hardening in a number of ways.

First, HFI offers per-thread memory-permissions which can be set without system-calls; thus a JIT compiler running on one thread can enforce a write permission on memory, while code compiled by a JIT executing on a separate thread can enforce an executable permission, restoring the benefits of write-xor-execute in a JIT setting.

Next, HFI can eliminate the limitations of JIT sandboxes such as Ubercage, by eliminating both the SFI overheads it suffers from today, and allow it to efficiently add new restrictions such as efficiently enforcing Control Flow and Spectre isolation—thus, eliminating the paths to bypass sandboxing that currently exist[12].

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[8] such as Memory Protection Keys, consistently fall short with respect to scaling, performance, security[9], etc. While 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 hardware support for in-process isolation that is both capable and easy to adopt.

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[4]; and providing safe extensibility in numerous applications. Beyond its use in Wasm, SFI is also used by Chrome to isolate it's JavaScript JIT; and by the Linux kernel (eBPF) to provide extensibility, performance analysis, and security monitoring.

The 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—while also opening the door to use cases that are simply not possible with SFI today.

To futher ease adoption, 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.

Finally, 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.

References

1. G. Tan, “Principles and implementation techniques of software-based fault isolation,” *Foundations and Trends in Privacy and Security*, 2017.
2. R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, “Efficient software-based fault isolation,” in *SOSP*, 1993.
3. R. M. Norton, “Hardware support for compartmentalisation,” tech. rep., 2016.
4. 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.
5. L. Clark, “Wasmtime reaches 1.0: Fast, safe and production ready!.” Bytecode Alliance. Accessed: Mar. 1, 2024. https://bytecodealliance.org/articles/wasmtime-1-0-fast-safe-and-production-ready, Sept. 2022.
6. 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.
7. 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*, pp. 1433–1450, 2021.
8. 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.
9. R. J. Connor, T. McDaniel, J. M. Smith, and M. Schuchard, “PKU pitfalls: Attacks on PKU-based memory isolation systems,” in *USENIX Security*, 2020.
10. 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, 2023.
11. B. Lord, “The urgent need for memory safety in software products.” CISA (.gov). Accessed: Mar. 1, 2024. https://www.cisa.gov/news-events/news/urgent-need-memory-safety-software-products, Sept. 2023.
12. S. Groß (saelo), “V8 sandbox - hardware support.” Accessed: Mar. 1, 2024. 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. Narayan received his doctorate degree from the University of California, San Diego. Contact him at shr@cs.utexas.edu.

Tal Garfinkel is a research scientist at the University of California, San Diego, San Diego, CA, 92093, USA. His research interests include hardware/software systems from programming languages to computer architecture, focusing on security, reliability, and performance. Garfinkel received his doctorate degree from Stanford University. Contact him at talg@cs.ucsd.edu.

Mohammadkazem Taram is an assistant professor

at Purdue University, West Lafayette, IN, 47907, USA. His research interests include computer architecture and computer security, focusing on microarchitectural attacks, high-performance mitigations, and architecture support for security and privacy. Taram received his doctorate degree from the University of California, San Diego. Contact him at kazem@purdue.edu.

Joey Rudek is a Ph.D. student at University of California, San Diego, San Diego, CA, 92093, USA. His research interests include 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, Mountain View, CA, 94043, USA. His research interests include computer and hardware security, including microarchitectural vulnerabilities, side-channel cryptanalysis, and security architecture. Moghimi received his doctorate degree from Worcester Polytechnic Institute. Contact him at danielmm@google.com.

Evan Johnson is a Ph.D. student at University of California, San Diego, San Diego, CA, 92093, USA. His research focuses on interests include software security and software correctness, particularly securing and verifying software sandboxing systems. Contact him at e5johnso@ucsd.edu.

Chris Fallin is a principal software engineer on the WebAssembly team at Fastly, San Francisco, CA, 94107, USA. His research interests include compilers, runtimes, and system software with an emphasis on correctness. Fallin received his doctorate degree from Carnegie Mellon University. Contact him at chris@cfallin.org.

Anjo Vahldiek-Oberwagner is a research scientist with the Datacenter Security Group, Intel Labs, Berlin, Berlin, 14167, Germany. His research interests include analyzing and building secure software and hardware computing systems, focusing on the usability of in-process isolation in data center workloads. Vahldiek-Oberwagner 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, Hillsboro, OR, 97124, USA. 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 the technical staff at Rivos Inc., Santa Clara, CA, 95054, USA, where he leads the platform security architecture and ISAs, and vice chair of the Security Horizontal Committee at RISC-V International. His research interests include exploit prevention instruction set architectures and confidential computing. Sahita received his M.S. degree in computer science from Iowa State University. Contact him at ravi@rivosinc.com.

Dean Tullsen is a professor at the University of California, San Diego, San Diego, CA, 92093, USA. His research interests include computer architecture and architectural security. Tullsen received his doctorate degree from the University of Washington. He is a fellow of the Association for Computing Machinery and a Fellow of IEEE. Contact him at tullsen@ucsd.edu.

Deian Stefan is an associate professor with the University of California, San Diego, San Diego, CA, 92093, USA. His research interests include principled and practical secure systems using techniques that span security, programming languages, and systems. Stefan received his doctorate degree from Stanford University. Contact him at deian@cs.ucsd.edu.