Segue & ColorGuard: Optimizing SFI Performance and Scalability on Modern Architectures

Shravan Narayan
UT Austin
Austin, USA

Tal Garfinkel
UC San Diego
San Diego, USA

Evan Johnson
UC San Diego
San Diego, USA

Zachary Yedidia
Stanford University
Stanford, USA

Yingchen Wang
UC Berkeley
Berkeley, USA

Andrew Brown
Intel
Hillsboro, USA

Anjo Vahldiek-Oberwagner
Intel Labs
Berlin, Germany

Michael LeMay
Intel Labs
Hillsboro, USA

Wenyong Huang
Intel
Beijing, China

Xin Wang
Intel
Beijing, China

Mingqiu Sun
Intel
Hillsboro, USA

Dean Tullsen
UC San Diego
San Diego, USA

Deian Stefan
UC San Diego
San Diego, USA

Abstract

Software-based fault isolation (SFI) enables in-process isolation through compiler instrumentation of memory accesses, and is a critical part of WebAssembly (Wasm). We present two optimizations that improve SFI performance and scalability: *Segue* uses x86-64 segmentation to reduce the cost of instrumentation on memory accesses, e.g., it eliminates 44.7% of Wasm's overhead on a Wasm-compatible subset of SPEC CPU 2006, and reduces overhead of Wasm-sandboxed font rendering in Firefox by 75%; *ColorGuard* leverages memory tagging (e.g., MPK), to enable up to a 15× increase in the number of Wasm instances that can run concurrently in a single address space, improving efficiency for high scale server-side workloads. We also explore the challenges of deploying these optimizations in three production toolchains: Wasm2c, WAMR and Wasmtime.

CCS Concepts: **• Security and privacy → Hardware security implementation**; **Browser security**.

Keywords: SFI, Wasm, sandboxing, optimization

ACM Reference Format:
Shravan Narayan, Tal Garfinkel, Evan Johnson, Zachary Yedidia, Yingchen Wang, Andrew Brown, Anjo Vahldiek-Oberwagner, Michael LeMay, Wenyong Huang, Xin Wang, Mingqiu Sun, Dean Tullsen, and Deian Stefan. 2025. Segue & ColorGuard: Optimizing SFI Performance and Scalability on Modern Architectures. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1 (ASPLOS '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3669940.3707249

This work is licensed under a Creative Commons Attribution International 4.0 License.

ASPLOS '25, March 30–April 3, 2025, Rotterdam, Netherlands.
© 2025 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0698-1/25/03
https://doi.org/10.1145/3669940.3707249

1 Introduction

Software-based fault isolation (SFI) [101] enforces isolation using compiler instrumentation in place of traditional hardware protection (e.g., page tables); thus, it avoids many of the overheads that processes and VMs impose [45, 75], such as high start-up times and expensive context switches. However, current SFI techniques impose limitations on performance and scalability. We introduce Segue and ColorGuard, two optimizations that mitigate these limitations by uniquely leveraging modern hardware.

Our optimizations were inspired by challenges faced in production systems built on WebAssembly (Wasm) — which critically relies on SFI. In recent years, Wasm has become an essential part of the software ecosystem: billions use Wasm daily in the browser [26, 29, 95, 96]; it provides extensibility in datacenter infrastructure [79], SaaS applications [25], and databases [91]; it mitigates memory safety vulnerabilities through libraries sandboxing [35, 72]; and it is a key enabler for high-scale low-latency edge-compute platforms from Fastly [78], Cloudflare [57], Akamai [1], etc.

While Wasm's success has led to a renaissance in SFI technology, it has also brought greater attention to its limitations: *Efficiency* — SFI overheads of 20%–30% or more are common [55, 93, 111] — this impacts Wasm's performance across many classes of systems, and also limits its adoption. For example, Firefox depends on Wasm to sandbox potentially vulnerable third-party libraries [35, 72], but cannot sandbox many media-decoding today libraries due to the user-visible delay it would introduce [28]. *Scaling* — current state-of-the-art SFI techniques waste large amounts of virtual address space, limiting the number of concurrent Wasm instances per-process [74]. Edge computing platforms are already hitting Wasm's scaling limits [77] — forcing them to rely on multi-process architectures that sacrifice throughput and latency, and increase software complexity (§2). Such scaling limitations only become more acute as application complexity increases [100]. Segue and ColorGuard address these limitations in the following way.

With *Segue* (§3.1), we note that SFI uses a base-and-bounds model of memory protection [62], similar to segmented memory systems. By leveraging the vestiges of segmentation support in the x86-64 ISA [50] (§3.1), we eliminate the cost of adding the "base", i.e., selecting the protection domain to access — by substituting segment-relative-addressing in place of the standard address computation used by SFI. This small change significantly reduces SFI overhead, cutting the number of instructions required for sandboxing memory operations in half, and also reduces register pressure. As a result, Wasm's overhead vs. native execution is reduced by 44.7% on SPEC CPU 2006, as well as 75% and 68% for sandboxing font rendering and XML parsing in Firefox (§6.1).

With *ColorGuard* (§3.2), we note that production Wasm implementations rely on large regions of mostly unused address space to catch out-of-bound memory accesses. While this avoids adding expensive bounds checks to every memory access, it also wastes most of the virtual address space. We can pack additional Wasm instances into this wasted space, while still maintaining isolation, by using memory tagging [2, 7, 8, 50] (e.g. Intel MPK), to assign each instance its own unique tag (color). This increases the number of concurrent Wasm instances a process can support by up to 15×, from the current limit of 16K instances to 256K instances. In comparison to using multiple processes for scaling, we found ColorGuard improves throughput by up to ≈ 29% in benchmarks that simulate FaaS edge platform workloads (§6.4).

We have upstreamed our optimizations into three industry supported Wasm toolchains: Wasm2c (Google), WAMR (Intel), Wasmtime (Fastly).

To the best of our knowledge, Segue is the first application of x86-64 segmentation in a production SFI tool, and ColorGuard is among the first production applications of MPK. Notably, unlike prior systems where scaling is limited by MPK [10, 45, 59, 82, 98, 99], ColorGuard illustrates how MPK can improve scaling.

While deploying these changes, we encountered a variety of challenges. For example, even after our ColorGuard code was upstreamed (code reviewed, fuzzed, etc.), we still weren't confident in its correctness. Specifically, ColorGuard modifies the address space layout in Wasmtime, which represents an explicit contract between the runtime and compiler. If this is incorrect, it can break isolation. Such bugs are the most common source of CVE's in Wasmtime [22, 23, 30, 44, 81]. Consequently, we formally verified our implementation (§5.2), revealing a bug, and several missing safety checks, for which fixes are being upstreamed.

We discuss how SFI is implemented in Wasm, and its performance and scaling limitations in §2. We explore how Segue and ColorGuard address these limitations (§3), and the challenges we encountered deploying these optimizations in real toolchains (§4 and §5). We then evaluate the performance improvements Segue and ColorGuard offer in three Wasm toolchains, and in an x86-64 SFI implementation based on LFI [109] (§6). Next, we then explore how ColorGuard can be implemented with memory tagging technologies (MTE [7], POE [40]) on ARMv9 (§7). Finally, we survey other work on SFI performance and scaling (§8).

2 Background and Motivation

Software-based Fault Isolation (SFI) [101] works by interposing on all memory accesses using compiler instrumentation. Using this mechanism, it virtualizes a process' address space into multiple isolated sandboxes. Because SFI eschews traditional hardware protection, it can offer unique properties that existing hardware-based isolation cannot.

These unique properties are core to what makes Wasm [43] — a platform-independent byte code that uses SFI for memory virtualization — compelling. For example, Wasm can rapidly switch between sandboxes at user-level — supporting IPC and context-switches that are as fast as function calls [60]; it can also create new sandboxes in microseconds (e.g. 5 µs in Wasmtime [31]) — orders of magnitude faster than creating processes or VMs [33, 45, 67, 75, 90]. These unique capabilities enable many novel use cases [1, 25, 26, 29, 35, 57, 72, 78, 79, 91, 95, 96] that would not be possible without SFI.

Unfortunately, SFI is often also Wasm's Achilles' heel. Memory access is on the critical path of many applications, and the added overheads of SFI instrumentation (often 20%–30% or more [55, 93, 111]), at best limit performance, and at worst render Wasm unusable. As mentioned, Wasm's overhead limits which libraries Firefox can sandbox [28], and there are doubtless many other applications where Wasm's many benefits are not outweighed by the performance tax it imposes vs native execution. To explore this, we start by reviewing how SFI is implemented in Wasm.

How SFI works in Wasm. SFI divides process address space into separate memory regions. Separation is enforced by an SFI compiler that modifies memory operations (load/store)

```
typedef struct {
    u64 dummy;  // offset: 0x0
    u32 arr[5]; // offset: 0x8
} foo;
 Pattern 1: Int to ptr, then deref 
// 64-bit int to be converted
u64 val = ...;
// convert to pointer and deref
u64 a = *(u64*)val;
 Pattern 2: Read arr elem in struct 
foo* obj = ...; // struct with arr
u32 idx = ...; // index to read

u32 b = obj->arr[idx];
```

(a) Example code snippet

```
; SFI performs mem ops as
;   heap_base + 32-bit offset
; heap_base=>rax, val=>rbx
; obj=>rcx, idx=>rdx
 Pattern 1 
; val = (u32)val;
mov ebx, ebx
; Load (heap_base + val)
mov r10, [rax + rbx]
 Pattern 2 
; tmp = (u32)&(obj->arr[idx])
lea edi, [ecx + edx*4 + 0x8]
; Load (heap_base + tmp)
mov r11, [rax + rdi]
```

(b) SFI compilation

```
; SFI performs mem ops as
;   heap_base + 32-bit offset
; heap_base=>gs, val=>rbx
; obj=>rcx, idx=>rdx
 Pattern 1 

; Load (heap_base + (u32)val)
;   in a single instruction
mov r10, gs:[ebx]
 Pattern 2 
; Load (heap_base +
;       (u32)&(obj->arr[idx]))
;   in a single instruction
mov r11, gs:[ecx + edx*4 + 0x8]
```

(c) SFI compilation with Segue

Figure 1. Segue in Practice: *This illustrates how two code patterns compile more efficiently with Segue: an integer-to-pointer conversion followed by a dereference, and reading an array element inside a struct. All pointer accesses are converted to the form "linear memory base (i.e., heap base) + a 32-bit offset" for sandboxing. Without Segue, each pattern takes two instructions, and uses* `rax` *to store the heap base. With Segue each pattern takes one instruction, frees up* `rax` *and an operand slot. Note that in x86, any 64-bit register (e.g.,* `rbx`*) is truncated to 32-bits, if an instruction uses its 32-bit variant (e.g.,* `ebx`*) as an operand.*

to ensure accesses fall within the region dedicated to the appropriate sandbox, and traps accesses outside that region.

Conceptually, an SFI compiler views the operand of each memory operation as an offset into a region, and enforces isolation in two steps: (a) it adds the starting address of current region to the offset (b) it adds a bounds check to ensure the operand is still in the appropriate region. In practice, bounds checks are often too expensive [69, 111] [1]. Thus, production Wasm engines instead enforce bounds implicitly using virtual memory, and fall back to bounds checks only when needed (in embedded domains without virtual memory, or 32-bit address spaces where virtual address space is limited).

To do this, Wasm engines allocate 4GB for each memory region (*linear memory* in Wasm terminology), followed by 4GB of unmapped memory (a guard region), for a total of 8GB per-sandbox. Wasm defines load/store instructions as taking two 32-bit unsigned operands. Thus, when a Wasm compiler generates code it adds these operands, resulting in a 33-bit address. It then takes this address and adds it to a 64-bit *base address* (the starting address of a sandbox's memory). The result of this addition then is used to perform the load or store. Consequently, addresses are always within 33 bits (8GB) of the start of linear memory by construction, and all memory accesses either hits linear memory (first 4GB), or the guard region (second 4GB) which traps. Some Wasm compilers like Wasmtime additionally implement optimizations to reduce guard regions from 4GB to 2GB (See §5).

Scaling in Wasm. FaaS edge computing platforms [1, 27, 57, 78] spin up new Wasm instances in microseconds [31], on every network request. Similar to other web services [85, 92], these requests often wait on IO from caches, microservices, etc. resulting in lots of outstanding concurrent requests.

Unfortunately, as each instance consumes 8GB (2^{33}) of address space, a server process can handle only a limited number of concurrent requests before address space is exhausted. Concretely, x86-64 CPUs provide a 48-bit address space[2], with only 47 bits available in user space [50] which means we can run at most 16K ($2^{47}/2^{33}$) Wasm instances per-address space. By the standards of modern web services [20, 92], 16K concurrent requests is not much, and edge providers have been hitting this limit for years [71, 77]. Interestingly, most of the 8GB allocated per-instances is never used — Wasm instances in FaaS settings rarely exceed a few hundred megabytes [27] leaving the 4GB linear memory largely unused, while the 4GB guard region is also dead space. As noted, Wasmtime's optimizations to reduce the guard regions to 2GB marginally increase this limit to roughly 21K, but still leaves a lot of unused space.

To address this concurrency limit, FaaS vendors resort to spinning up more processes to improve scaling and avoid under-utilizing CPUs. However, this introduces several problems. First, going multi-process adds context-switch overheads and scheduling delays, increases cache contention, and introduces load imbalances — all of which hurt performance. Next, FaaS applications don't always consist of a single function, and when functions communicate across processes, it is 1000× to 10000× slower [45, 60]. Finally, going multi-process adds complexity, making it harder for platform developers to build new features and diagnose performance issues [27].

[1]Historically, bounds were also enforced with masking [101]. However, this causes out-of-bounds loads/stores to wrap around. This results in memory corruption, rather than a deterministic traps as Wasm requires.

[2]A small fraction of CPUs support 52/57-bit address spaces, however, using this expanded address space presents additional challenges (§8).

Growing hardware capacity, greater serverless adoption, and changes to Wasm that increase the number of memories per-application [100] will only exacerbate these challenges.

3 Segue and ColorGuard

We present Segue and ColorGuard, two optimizations for SFI performance and scalability, respectively.

3.1 Reducing SFI overhead with Segue

Segue leverages the vestiges of segmentation in the x86-64 ISA to reduce SFI overhead for memory operations. By way of context, segmentation is deeply embedded in the x86-32 memory model [21], and was a key enabler for both academic [34] and production [110] SFI systems in the past.

However, popular OSes made little use of segmentation, leading AMD to remove most support when they introduced x86-64. x86-64 retains just two segment registers — `%fs` and `%gs`, with minimal functionality. In particular, segment limit checks were eliminated. As segments could no longer enforce limit/bounds checks on memory operations, they were replaced by other mechanisms and were no longer perceived to be useful for SFI [24, 86]. With Segue, we show that the vestiges of segmentation in x86-64 are still quite useful.

Segue optimizes SFI memory access, by storing the base address of the current memory region being accessed in a segment register, then using segment-relative addressing for memory operations. Figure 1a, illustrates this in two common patterns for accessing memory: an integer-to-pointer conversion followed by a dereference, and an access of an array in a struct. Figure 1b shows the (simplified) assembly code that current Wasm/SFI tools generate for this code, and Figure 1c shows the same code compiled using Segue.

Without Segue, each memory access requires computing the heap-base (stored in register `%rax`) plus a 32-bit offset to ensure isolation (the SFI address space is assumed to be 4GB). With Segue, the heap base is stored in the `%gs` register, and is used directly in the `mov`, with segment relative addressing. Additionally, without Segue, we see that each code pattern compiles to two instructions; but, with Segue, compiles to one instruction. These changes allow Segue to improve performance because it:

Frees an operand slot in mov instructions — x86 supports a variety of memory addressing modes that take several operands, to support complex address calculations. Usually, SFI/Wasm compilers must reserve one of these operand slots for a linear memory base address, as shown in lines 9 and 14 in Figure 1b (the base address is stored in register `%rax`). Thus, the compiler cannot use this operand slot for other inline addition. However, since segment relative addressing does this addition as part of the load, Segue frees this operand slot for the compiler to use, as shown on line 14 in Figure 1c.

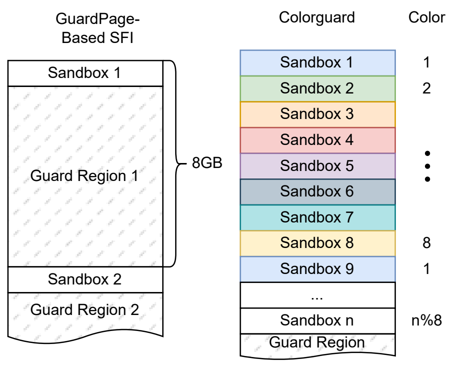


Figure 2. Scaling with ColorGuard: *an example of packing 8× more sandboxes as traditional guard-region based SFI, into a 8GB region using 1GB sandboxes. Each sandbox in the 8GB is assigned its own unique MPK color (tag). Colors are allocated (striped) across memory to ensure identically colored sandboxes are always 8GB away from each other.*

Reduces instructions by allowing mixed-mode arithmetic — SFI/Wasm compilers calculate machine addresses using 64-bit arithmetic even though indexes into linear memory are 32-bits. This is because adding a 64-bit base address to the 32-bit offset is not permitted in a single instruction in x86. However, with Segue, such mixed-mode arithmetic is possible by using the address-size override prefix. As shown on line 14 in Figure 1c, this allows SFI compilers to replace two instructions with one.

Frees up a general-purpose register As discussed, Segue also frees up a general-purpose register (GPR). We see that `%rax` is used for the linear memory base address before Segue (Figure 1b), and `%gs` being used after (Figure 1c). This frees up `%rax` for other computations.

Other considerations. For practical deployment, a few details bear consideration. To start, OSes dedicate one segment register to thread-local storage (TLS) (e.g., `%fs` on Linux); thus, one segment register is free for other uses [58]. Next, while all x86-64 CPUs support Segue, CPUs since IvyBridge (2011) offer userspace instructions to modify segment registers (`wrfsbase`, etc.) in lieu of expensive system calls; this is important for fast context switches. Finally, segments will remain a stable part of x86-64 [49]; as removing/modifying them would break all existing x86-64 binaries that use TLS.

While often a significant performance win, Segue also has some costs. Using the segment prefix and the address-size override prefix slightly increases the length of memory instructions. Also, there is a runtime cost to setting the `%gs` register to the heap base of a new region when context switching into a Wasm instance — although this is quickly amortized. In §4, we touch on other practical considerations when deploying Segue in production toolchains, and show how the benefits of Segue far outweigh these costs in §6.

3.2 Improving Scalability with ColorGuard

ColorGuard increases Wasm scalability, i.e., the number of instances that can run concurrently in each process address space by up to 15×, based on two observations: (1) As discussed (§2), a Wasm instance's 8GB allocation (4GB address space + 4GB guard region), is mostly unused space. By definition, the guard region is unmapped memory, and many Wasm FaaS workloads rarely exceed more than a few hundred MB [27], (2) We can safely repurpose the unused space for (smaller) linear memories, if other executing Wasm instances cannot access this repurposed space.

To do this, we leverage Memory Protection Keys (MPK)[3] [2, 50], a feature that allows applications to control access to memory via. keys — to enforce the constraint that memory not used by the currently running sandbox is inaccessible. The key (also called the color) is a 4-bit value that is stored in page table entries (PTEs). Once an application has assigned colors to pages via system calls (i.e., `pkey_mprotect()`), it specifies which color the current thread can access in userspace in the `pkru` register. As `pkru` updates are fast (≈ 40 cycles (§6)), a thread can rapidly change accessible colors.

Using this mechanism, ColorGuard stripes memory so that sandboxes within an 8GB range have different colors, and sets the `pkru` register so the current thread can access only the color of the active sandbox. Other process memory, such as memory being used by a Wasm runtime, is completely unchanged and assigned the default MPK color (0).

Figure 2 contrasts the ColorGuard striping pattern with traditional guard regions for sandboxes with 1GB linear memories and 7GB guard regions. Here, we see that ColorGuard requires every sandbox that occupies memory in the 8GB following a given sandbox to use a different color for its linear memory. Specifically, MPK colors stripe the 7GB region following the end of sandbox 1 — offering an 8× increase in sandbox density. In our example, any out-of-bounds memory access from sandbox 1 would trap as it would hit a region with a different color. We could further increase density to 15×, by using all of MPK's colors and creating smaller sandboxes, i.e., for sandboxes of $8GB/15 \approx 550MB$ — though in practice we find real allocators impose additional constraints (§5.1), that allow still smaller ≈ 400MB sandboxes (§6.4).

This striping pattern scales up to any number of sandbox chains — sandboxes that are placed in adjacent memory regions. We only need guard regions in a sandbox chain in two instances: (1) after the final sandbox to ensure the last sandbox in the chain is protected, and (2) if 15 consecutive sandboxes use less than 8 GB combined, we'll need a guard region before using the first color again. A Wasm runtime could also potentially chain sandboxes of different sizes to efficiently use colors and possibly eliminate the second case.

[3] MPK appeared in Skylake (2017) and Comet Lake (2019) in Intel server and client CPUs respectively, and in EPYC Milan (2021) in AMD CPUs. ARMv9 offers this functionality with its permission overlay extension (POE) [9, 40].

Finally, we note that prior sandboxing systems that rely on MPK exclusively for isolation have faced security challenges such as controlling access to potentially unsafe instructions or system calls that could bypass MPK [19, 82, 99]. However, such issues are not relevant to Wasm, as Wasm compilers, by definition, control which instructions are emitted and Wasm runtimes don't allow direct access to system calls; rather Wasm code can only call higher level interfaces such as WASI [103], that don't invoke these unsafe system calls. Additionally, MPK prevents speculative access to pages with non-permitted colors [50, 53] and thus offers similar guarantees to guard regions.

In the next two sections, we discuss implementing and upstreaming Segue and ColorGuard in SFI toolchains.

4 Implementing Segue

We discuss our experience developing and upstreaming Segue in two production Wasm toolchains (Wasm2c, WAMR), and in one research SFI system (LFI [109]), and explore the practical challenges we encountered.

4.1 Implementing Segue in Wasm2c

Wasm2c [94] is a transpiler that transforms Wasm to a limited subset of C, offering the benefits of Wasm isolation in a form that is easy to compile and link with existing tools. Wasm2c is part of the Wasm binary toolkit (Wabt), a widely used set of Wasm tools maintained by Google, and is used by Firefox to mitigate memory safety vulnerabilities by sandboxing third-party C libraries [46, 72].

We modified Wasm2c so that accesses to Wasm linear memory are performed through a segment register. For this, we used a GNU-extension called named address spaces [37] that allows pointers in C to indicate that they belong to a particular segment. We then augmented the Wasm2c runtime to execute instructions that set the segment base on x86 using compiler intrinsics. Finally, we compiled the emitted C code and Wasm2c runtime with Clang to produce our binary.

Our initial implementation was pleasantly simple. It works directly with the C compilation pipeline which allows C compilers to fully take advantage of all of the benefits of Segue (§3.1) — the extra register, the extra addressing operand, and the inline truncation — with no extra effort, and thus allowed us to evaluate the performance benefits of Segue (§6.1). As we discuss in §4.2, such complete integration is far more involved in other toolchains.

Bringing Wasm2c to production. Turning our prototype into production ready code required two changes. First, to make Segue work cross-platform required adapting it to a variety of different OSes and C compilers. We also needed to test for the presence of x86-64 segmentation and gracefully fall back if not available.

Next, we needed to add support for switching the segment register (linear memory base address) at the appropriate time — Wasm2c supports multiple modules, and each may have more than one memory — a Wasm feature called multi-memories [5]. To enable this, we choose a simple design that avoids the need for Wasm2c developers to worry about tracking and resetting this register, which is error prone and could compromise isolation. Instead, we modified the Wasm2c runtime to set the segment base on function entry when called from *outside* of the current Wasm module, i.e., when entering the module; while functions called from within the same module use an alternate code path that elides the setting of this register. This approach thus minimizes redundant register assignments, while maintaining simplicity.

Segue has been upstreamed into Wasm2c, which will be used to enable Segue in Wasm-based library sandboxing used by production Firefox [72]. Firefox adds another level complexity as it is run both on new CPUs as well as older CPUs that do not support the FSGSBASE extension (user level access to `%FS/%GS`). Thus, we must gracefully fall back to setting these registers using system calls (e.g. `arch_prctl()`), and carefully account for the added overhead this entails.

As we describe next, integrating Segue into more complex toolchains like WAMR is more involved.

4.2 Implementing Segue in WAMR

WAMR [48] is a standalone WebAssembly compiler and runtime that supports ahead-of-time (AOT) and JIT compilation, developed by Intel. Unlike Wasm2c, it uses the LLVM compiler backend — converting Wasm to LLVM IR, and relying on LLVM to convert this to native instructions. WAMR supports a wide range of platforms including embedded devices and trusted execution environments (e.g., Intel SGX).

WAMR has shipped a (limited version of) Segue for over a year [51]. Initially, supporting Segue in WAMR seemed straightforward. We modified WAMR [6] to set the segment register and to emit memory access instructions using segments. Unfortunately, implementing the full (and simple) design from §3 revealed several challenges.

First, WAMR cannot easily make use of the extra operand slot discussed in §3.1. This would require substantial modifications to WAMR's code generation and optimization passes that would only benefit Segue users. Thus, the WAMR team opted to adopt a more limited form of Segue that only uses segment-based memory access to free a register (and leverages Segue's fast heap base addition). In practice, this means that WAMR does not always reduce the number of instructions emitted, like Wasm2c (Figure 1b). However, freeing up of a register and Segue's more efficient instruction encoding still lead to real world performance improvements.

Indeed, in our initial tests of Segue with WAMR's benchmark suite (discussed in Section 6.2), we saw performance improvements in most cases. However, we also saw some performance regressions in a few benchmarks: the `sieve` benchmark (prime number computation) and `memmove` (moving data between two buffers) from the Sightglass suite [15].

These regressions are not fundamental to Segue: WAMR includes a number of Wasm-specific vectorization passes (e.g., such as converting long load sequences, loops, etc. to SIMD instructions [52]) that make assumptions about the generated code. Still, changing these optimization passes to make them Segue-aware is not trivial; the straightforward approach would make them less platform-neutral, hindering WAMR's ability to support the many instruction sets it does today. We thus only expose Segue behind a flag that the user can selectively enable; in practice, this also allows users to selectively tune the optimization to their workload (e.g., by enabling Segue selectively on loads-only or stores-only) and, indeed, we were able to address the regressions in-turn.

4.3 Segue in LFI

To explore the implications of Segue in a state-of-the-art SFI system that is not Wasm, we implemented Segue in LFI [109]. LFI was originally developed for ARM64, and works by rewriting assembly code to insert SFI instrumentation. We implemented an x86-64 backend for LFI (with and without Segue) in 700 lines of code, using isolation techniques from NaCl [86] for sandboxing loads, stores, and jumps.

While our implementation was mostly straightforward, a key difference between Segue with Wasm vs. LFI is that we still need to reserve a GPR for the region base address — but not for isolating memory operations, rather for control-flow.

LFI (like Wasm) relies on an entirely different scheme for control-flow isolation than Wasm. Unlike Wasm, LFI restricts code to the 4GB sandbox region and thus both forward (indirect jumps) and backward edge control-flow (return instructions) have to be bounded to this region. Unfortunately, we cannot use the x86-64 segment registers (`%fs/%gs`) on control-flow targets; rather we fall back to SFI's default approach — using explicit instructions to add the region base (stored in a GPR) to the return address restricted to 32 bits. Finally, due to the large number of control-flow instructions in a typical binary, spilling/reloading this GPR is not viable.

5 Implementing ColorGuard

We implemented and upstreamed ColorGuard in Wasmtime, an open source Wasm runtime developed by multiple industry contributors, used in SaaS, serverless, and edge computing settings by Fastly, Microsoft Azure, Shopify, and others [31, 78]. Because it is designed to support demanding server-side workloads, Wasmtime is far more sophisticated than other Wasm runtimes. Also, Wasmtime is tightly integrated with a custom compiler backend, Cranelift. Thus, unlike the other SFI toolchains we evaluated, the Wasmtime/Cranelift developers own every step of compilation. This added complexity, and the scope of Wasmtime's production use required additional rigor in testing and deploying our changes, including formally verifying the correctness of our implementation (§5.2).

Num	Wasmtime invariant	Goal
1	`total_slot_bytes == pre_slot_guard_bytes + slot_bytes * num_slots + post_slot_guard_bytes`	Ensure there are no memory leaks by checking that piecemeal allocating slots amount to the total allocation.
2	`slot_bytes >= max_memory_bytes`	Each slot must be large enough for the Wasm memory to grow into.
3	`is_page_aligned(slot_bytes, max_memory_bytes, pre_slot_guard_bytes, post_slot_guard_bytes, total_slot_bytes)`	Enforce alignment on page-related parameters especially since even minor divergence could can break memory isolation.
4	`num_stripes <= num_pkeys_available && num_stripes <= num_slots && num_stripes >= 1`	Ensure that we use at most many stripes (and slots) as MPK supports.
5	`num_stripes <= (guard_bytes/max_memory_bytes) + 2`	Ensure minimum number of stripes (and thus protection keys). If the next MPK-protected slot is bigger or the same as the required guard region, we only need two stripes; otherwise if the next slot is smaller than the guard region, we need enough stripes to add up to at least that guard region size.
6	`bytes_to_next_stripe_slot >= expected_slot_bytes.max(max_memory_bytes) + guard_bytes` `slot_bytes + post_slot_guard_bytes >= expected_slot_bytes`	Ensure that using stripes to scale the number of sandboxes doesn't introduce vulnerabilities like out-of-bounds access. To this end, we (1) may have reduced the slot size from `expected_slot_bytes` to `slot_bytes` assuming MPK striping; and, (2) enforce that the last slot doesn't rely on MPK for striping.
7	`expected_slot_bytes mod WASM_PAGESIZE == 0`	**[Missing]** Slots must be a multiple of Wasm's page size (64KB).
8	`max_memory_bytes mod WASM_PAGESIZE == 0`	**[Missing]** Max memory must be a multiple of Wasm's page size (64KB).
9	`expected_slot_bytes <= total_memory_bytes guard_bytes mod OS_PAGESIZE == 0`	**[Missing]** If pre-guardpages are used, it must be a multiple of OS page size (4KB).
10	`expected_slot_bytes <= total_memory_bytes`	**[Missing]** Total slots size should fit into usable memory.

Table 1. ColorGuard safety invariants in Wasmtime.: *Invariants 1-6 show initial checks upstreamed by the Wasmtime team. Invariants 7-10 shows missing checks that our verification effort revealed that could permit invalid/unsafe configurations.*

5.1 Implementing ColorGuard in Wasmtime

As discussed in §3.2, implementing ColorGuard consists of three steps: (1) obtaining MPK colors at startup; (2) striping (coloring) memory in Wasmtime's allocator; (3) changing colors during transitions into or out of a sandbox. Next, we describe each step, then discuss other deployment challenges.

Obtaining protection keys. MPK allows a user level program, i.e, Wasmtime, to enforce page permissions using up to 15 keys/colors. To use these, Wasmtime uses the system call `pkey_mprotect()` to assign a color to pages and configures the `pkru` register to specify which colors (and thus pages) can be accessed by the current thread. As some platforms may not support MPK, we modified Wasmtime to check for its availability. We also added a user configurable parameter to specify how many keys are available, in case Wasmtime is used in an application that uses some keys for other purposes. Wasmtime could also dynamically infer the free key count via error codes from the `pkey_alloc()` syscall.

Striping Memory in Wasmtime's Allocator. We extended Wasmtime's pooling allocator to support ColorGuard. The allocator pre-allocates a large slab of memory (the pool) at application startup using `mmap()` — and then splits it into *slots*, delineated by guard regions, that will be used as linear memories. When a Wasm instance finishes execution, the allocator zeroes the memory in the slot with the `madvise()` syscall, so that it can reuse this slot for a new Wasm instance.

Unlike our earlier description of address spaces and guard regions, the allocator does not exclusively support a 4GB address space + 4GB guard region memory layout; instead, its layout is configurable. Both the address space size, guard region size, and their layout can change to support different bounds checking mechanisms (guard regions vs. bounds checks, or even a mix of the two), and memory organizations.

We mention this for two reasons. First, the details of memory layout are used to construct an explicit contract between the allocator and compiler. If this contract is not maintained, it will break isolation. Thus, when modifying the allocator to support ColorGuard, these details are important for correctness; we discuss this further in the next section on formally verifying our changes. Second, this added sophistication exists largely due to Wasmtime's requirements for scalability.

For example, the allocator, by default, does not employ the standard Wasm configuration of a 4GB address space followed by 4GB guard regions. Instead, it pads each Wasm instance with 2GB of *pre-guard region* and 2GB of *post-guard region* before and after the sandbox memory respectively. This modified scheme allows two Wasm instances in adjacent slots, A and B, to share 2GB of guard region — the post-guard region of the Wasm instance A can serve as the pre-guard region of Wasm instance B. Thus, instead of 8GB per-instance, only 6GB per instance is required, allowing roughly 20K instances per process, vs. the usual 16K.

This alternate memory layout is supported by modifications to how Cranelift generates code to access linear memory (§3.1). To wit — for sandboxes with a maximum memory limit less than 2GB, Cranelift implements linear memory base addition using a *signed* (rather than the normal unsigned) offset; thus any offset beyond 2GB would trap in the pre-guard region, as it is interpreted as a negative index.

To further complicate things, Wasmtime also supports guard regions of arbitrarily small sizes; in this scenario the compiler (Cranelift) employs explicit bounds checks except when it can statically prove that these limited guard regions are sufficient for safety.

The memory pool has to calculate its slot layout accounting for all of this. To facilitate this, it accepts as parameters: the number of slots, the maximum memory size, the size of the guard regions, and whether pre-guard regions should be included. Using this, it computes the memory layout, and returns a layout data structure describing how the pool slab will be divided up — it supplies this as a contract to the compiler (Cranelift), so it can generate code appropriately.

To support ColorGuard, we extended this layout computation to support striped memory. One part of this is determining how many colors we will need to stripe memory, i.e., to create a repeating cycle of unique colors. In the simple case, this is just the size of our guard regions (e.g. pre-guard + post-guard), divided by our slot size (this tells us how many linear memories can fit into the space used by our guard regions), plus one additional color for the slot those regions protect. For example, if we have 4GB of guard regions, and our slots are 2GB each, we need $(4/2) + 1 = 3$ colors.

However, the calculation must also handle the scenario where there are not enough keys available to create the stripes; here, we need to modify the calculations to use a combination of stripes and guard regions to maintain the contract with the JIT compiler. After this layout is calculated, in addition to providing it to the compiler, the allocator protects each stripe with `pkey_mprotect()`; as well as the appropriate guard regions with `mprotect(...,PROT_NONE)`.

Changing protection keys on context switch. Finally, we instrumented Wasmtime to set the accessible MPK stripe on each transition into and out of a Wasm instance. Aside from the fact that Wasmtime specializes transitions for a variety of contexts — sync vs. async transitions, function calls vs. jumps, etc. — this is mostly straightforward. We enable only the key for the current stripe during the transition in; and we disable this restriction when the Wasm instance calls back into the host runtime, for example, to invoke a system call via WASI [103] (Wasm's interface for system operations).

Other deployment considerations. Deploying MPK in Wasmtime involved dealing with several practical concerns. First, GitHub's infrastructure does not currently guarantee MPK support, which is a challenge for fuzzing and integration testing. While QEMU can be used to address this, it is yet-another-thing that needs to be understood and maintained by Wasmtime developers. Next, as each MPK-protected stripe creates a new virtual memory allocation (VMA) in the Linux kernel, the default kernel limit of 65K (`vm.max_map_count`) must be increased to fully utilize ColorGuard. Finally, MPK is rather esoteric for most Wasmtime users, thus, both documentation and detailed working examples were upstreamed to make this feature accessible.

5.2 Verifying ColorGuard

ColorGuard's implementation modifies one of the most security-critical parts of Wasmtime: the memory allocator. Bugs here are the most common source of CVEs in Wasmtime [22, 23, 30, 44, 81, 104] — because these bugs often break isolation.

To avoid such bugs, the Wasmtime team specified a set of invariants this new code should enforce when integrated with the existing memory pool allocator. Table 1 (Invariants 1-6) shows the invariants specified by the Wasmtime team, who implemented these as property tests that they then (tried to) check using their fuzzing infrastructure. While fuzzing did not reveal anything, the team was still concerned about bugs, which lead us to reach for formal verification.

Specifically, we translated the invariants to logical formulas that we then checked using the Flux [64] refinement type checker. Unlike the Wasmtime team, we did not make assumptions about the allocator's interface: We verified the memory allocator's correctness by specifying its types under a stronger attacker model — that the allocator is called with a potentially unaligned, unsafe, or otherwise incorrect inputs (and state). This ensures the allocator is defensive and can be updated without introducing new security bugs [4].

In total, we verified 133 lines of Rust using 34 lines of Flux annotations, and a 15 line Z3 proof for bitwise arithmetic that could not be checked in Flux. Our proof shows that address space isolation holds regardless of parameters passed to the ColorGuard-sandbox allocator API, assuming that the program respects Rust semantics. The proof required roughly two weeks for an experienced Flux and Z3 user and is checked in under a second.

Our verification effort confirmed the Wasmtime teams worries: we found one bug in the implementation and four new preconditions (Table 1, Invariants 7-10) that the allocator must satisfy for it to behave correctly (to specification).

The bug consisted of a saturating addition that should have been a checked addition: if the addition ever actually saturated, it would break the invariants in Table 1.

Of the missing preconditions, three are constraints on the alignment of inputs, one is a requirement that the result size is smaller than the total size of the allocation. While

[4]The allocator code does not change very often, so we do not plan to upstream the proof to avoid new dependencies on Wasmtime.

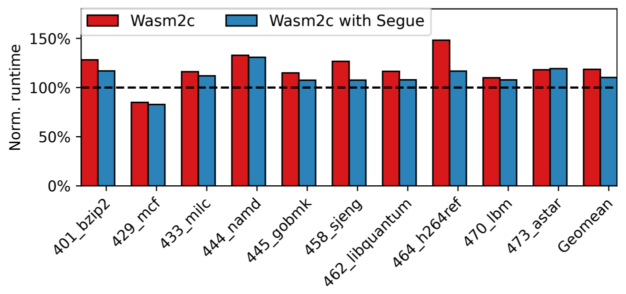


Figure 3. Segue on Wasm: *SPEC CPU 2006 on Wasm2c normalized to native code performance. Segue reduces the geomean by 8.3%, eliminating 44.7% of Wasm's overheads.*

	Wasm2c	Wasm2c with Segue	Size reduction
bzip2	484 KB	448 KB	7.4%
mcf	396 KB	384 KB	3.0%
milc	780 KB	692 KB	11.3%
namd	1036 KB	948 KB	8.5%
gobmk	4692 KB	4444 KB	5.3%
sjeng	652 KB	616 KB	5.5%
libquantum	468 KB	448 KB	4.3%
h264ref	1600 KB	1404 KB	12.3%
lbm	396 KB	388 KB	2%
astar	568 KB	532 KB	6.3%

Table 2. Compiled binary sizes of the SPEC benchmarks comparing stock Wasm and Wasm with Segue in the Wasm2c compiler. Segue decreases binary size by a median of 5.9%.

Wasmtime doesn't call the allocator with such unsafe values today, code that leaves defensive checks implicit typically end up being the source of security bugs — and indeed bugs in the allocator could be abused to break isolation.

The fix itself is straightforward, consisting of a few extra checks in the allocator; we are currently working to upstream these to ensure the runtime is formally guaranteed to enforce these invariants going forward.

6 Evaluation

We evaluated Segue and ColorGuard in the upstream WAMR (v1.3.2) and Wasmtime (main branch, Feb 8th 2024) respectively, and our fork of Wasm2c. Benchmarks are run on a desktop class Intel RaptorLake i9-13900KS (5.4 GHz, max 6GHz), on a performance core, with 128 GB of RAM, running Ubuntu 22.04.3 LTS, with hyper-threading disabled. Benchmarks are pinned to a single isolated CPU whose frequency is fixed at 2.2GHz. Unless otherwise noted, all reported benchmarks have a standard deviation of less than 1%.

6.1 Segue on Wasm2c

We evaluated Segue's impact on performance and binary size in Wasm2c with:

- **SPEC CPU 2006:** We choose SPEC CPU 2006 over SPEC CPU 2017 as its memory requirements often exceed Wasm 4GB limit. We also excluded several benchmarks that were not Wasm compatible (following Narayan et. al [73]).
- **Firefox's font rendering:** uses a font rendering library, `libgraphite` [41], that is sandboxed using Wasm to ensure that any memory-safety errors are contained. To measure performance, we recorded the time taken to reflow text on a webpage ten times with different font sizes and report the median. As Firefox uses separate invocations to render each letter/glyph on the webpage, this benchmark also captures the cost of setting the segment base prior to each invocation of a Wasm-sandboxed library.
- **Firefox's XML parsing:** also uses a library, `libexpat` [65], that is sandboxed using Wasm. To measure XML parsing performance, we repeat the benchmark used to evaluate the performance of `libexpat` when it was first sandboxed [14]. We use Firefox's built-in profiler to measure the load time of an SVG image (which is defined using XML) from a large website, Google Docs. The SVG contains the toolbar icons used by Google docs, and is concatenated 10 times to amplify the benchmark and reduce noise. We measure this cost 10 times and report the median.

Analysis. Segue offers marked speedups over the default compilation. Segue eliminates 44.7% of Wasm's geomean overheads in SPEC CPU 2006. Although not shown, it can also be applied to optimizing Wasm engines that use explicit bounds checks; here it eliminates 25.2% of the overheads. This matters because Wasm engines today rely on explicit bounds checks to support 64-bit Wasm memories [3, 24], that cannot leverage guard regions.

Segue also offers median reduction in binary size of 5.9% and a max reduction of 12.3% for SPEC CPU 2006, this follows from the fact that Segue reduces the number of instructions for memory access by half (§3.1).

For Firefox's font rendering, unsandboxed font rendering takes 264 ms. Sandboxed font rendering takes 356 ms without Segue, and 287 ms with Segue. Thus, Segue eliminates 75% of the overheads of sandboxed font rendering.

For Firefox's XML parsing, unsandboxed XML parsing takes 331 ms. Sandboxed XML parsing takes 381 ms without Segue, and 347 ms with Segue. Thus, Segue eliminates 68% of the overheads of sandboxed XML parsing.

Outliers. We observe a couple of outliers. First, we see `429_mcf` runs faster in Wasm than native; this is not entirely surprising — since Wasm pointers are 32-bit offsets rather than native 64-bit pointers, Wasm can act as cache optimization [89, 111] in certain programs.

Next, we see that `473_astar` is slightly slower with Segue. We believe this is due to the increased size of memory instructions when using the `%gs` prefix. While Segue generates smaller code overall, this may not hold true for the few instructions in the tight inner loop of some benchmarks. If this

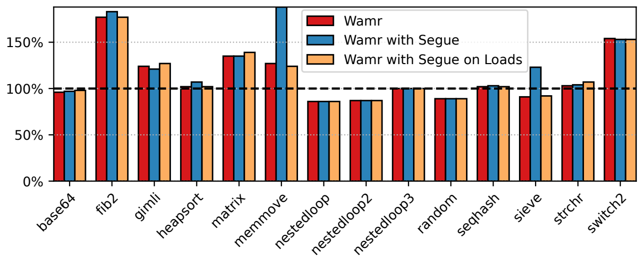


Figure 4. Sightglass performance on WAMR normalized to native code performance when Segue is used for (1) loads and stores, (2) loads only. Most differences are in the noise due to the small size of benchmarks. Segue shows some slowdowns due to engineering gaps in vectorization optimizations (§4.2), however, this is not fundamental. Segue for loads does not exhibit slowdowns.

expanded instruction size (and resulting instruction cache degradation) is not balanced by Segue's other performance benefits, the result is some performance overhead. To avoid these overheads, one could modify the compiler to choose between the Segue approach and an explicit base addition using a cost function. We leave this to future work.

6.2 Segue on WAMR

To evaluate WAMR's (limited) version of Segue, we used the benchmarks used by WAMR developers: PolybenchC [66], Dhrystone [106], and Sightglass [15]. We run these using the existing benchmarking scripts in the WAMR repo to ensure our evaluation is consistent with upstream practices.

PolybenchC contains applications from domains such as linear algebra, image processing, physics simulation, etc. Dhrystone is primarily tailored to estimating CPU performance, and is also used to test compiler optimizations. Sightglass was developed by the Bytecode Alliance (which develops tools for Wasm) and consists of common cryptographic and mathematical benchmarks, micro-benchmarks that test primitives like `memmove`, switch statements etc. Sightglass in particular includes benchmarks referenced in §4.2 that sometimes showed slowdowns.

Analysis. We compiled our benchmarks with Clang-17.0.6 for native code and WAMR for Wasm. Similar to §6.1, a few benchmarks in PolybenchC are faster in Wasm than in native resulting in a Geomean speedup of 6% for Wasm over native.

Segue further improves WAMR's performance resulting in a Geomean 10% faster than native. Dhrystone similarly runs faster in Wasm than native, improving performance by 9.7% over native. Segue improves this to further to 28.2% over native. For Sightglass (Figure 4), the performance changes in most of these benchmarks are in the noise due to their small size; however, the performance of two benchmarks `memmove` and `sieve` get slower by 35.6% and 48.7% respectively with Segue; we discuss this next.

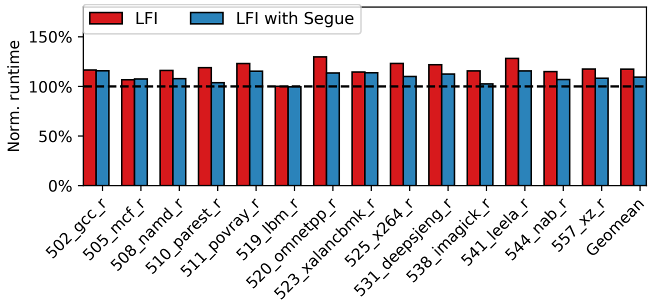


Figure 5. Segue on LFI: *SPEC CPU 2017 on LFI normalized to native code performance. Segue reduces the geomean by 8%, eliminating 46% of LFI's overheads.*

Outliers. As discussed in Section 4.2, Segue can sometimes interact poorly with WAMR's custom optimization passes. In this case, Segue results in the optimization pass not recognizing memory operations that could be vectorized, thus resulting in slower performance. WAMR's optimization pass can, in principle, be modified to recognize Segue to restore performance, although there may be practical engineering challenges to this (§4.2). We also found that employing WAMR's ability to tune Segue to apply only to loads eliminates these slowdowns; while not a perfect solution, this gives developers a way to use Segue selectively depending on their workload. Finally, we note that slowdown in benchmarks like `memmove` are unlikely to have a large impact on typical Wasm programs, as most programs use the memory manipulation functions in the C standard library — functions that directly use Wasm's vectorized bulk memory operations [4]

6.3 Segue on LFI

We evaluate the benefit of Segue for LFI on the SPEC CPU 2017 benchmark suite [13], using the same 14-benchmark subset as prior LFI work [109]: all SPECrate benchmarks that use only C/C++ and are compatible with musl-libc. We use GCC 13.2.0 with LTO enabled. The results are shown in Figure 5. Without Segue, the baseline SFI approach incurs a geomean overhead of 17.4% compared to native code. With Segue, this overhead is reduced to 9.4%, eliminating 46% of LFI's overhead. Interestingly, Segue offers remarkably consistent performance improvements across our Wasm2c and LFI benchmarks, even though the two rely on different SFI schemes, compilers, and different versions of SPEC.

6.4 ColorGuard on Wasmtime

We used a microbenchmarks and synthetic FaaS workloads to measure ColorGuard's overheads and scaling benefits.

6.4.1 Transition microbenchmark. When transitioning (context switching) in and out of Wasm instances (on invoking the WASI [103] API, switching between Wasm instances, etc.), transition code does a variety of tasks that

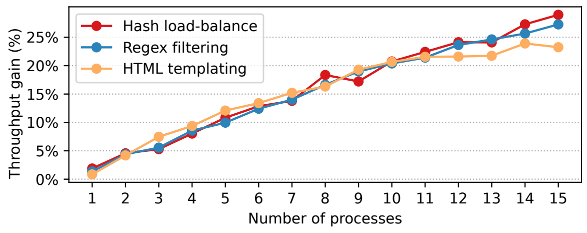


Figure 6. Multiprocess Scaling vs. ColorGuard: *Results from simulated FaaS workload on a single core. As the number of processes required to scale increases, ColorGuard offers more speedup, with a maximum of* ≈ 29%.

can include switch stacks, set exception handlers, adjusting for Wasm's ABI, etc. With ColorGuard, an additional instruction is added to switch MPK domains. We measured the impact of this added cost on different transitions [17] with 10 repetitions and report results below. On average, Wasmtime's per-transition cost increases from 30.34 ns to 51.52 ns, a roughly 20ns (44 cycle) increase. As transitions in Wasmtime are relatively infrequent, and are often used for expensive actions such as system calls, this added cost is generally amortized, as we see on our macrobenchmark.

6.4.2 Scaling microbenchmark. To begin, we exercised Wasmtime's scaling with a simple example program that instantiates Wasmtime's pool allocator (§5) with 408MB slots (linear memories). Without ColorGuard, we were able to to create 14,582 memory slots, and with and with ColorGuard, we could allocate 218,716, an increase of ≈ 15×.

6.4.3 ColorGuard Scaling vs. Multi-Process. To evaluate the benefits of ColorGuard vs. using multiple processes for scaling, we assembled a set of benchmarks typical of FaaS edge environments [18, 32] — HTML templating, hash-based load balancing, and regular expression filtering of URLs.

To reduce noise, and only measure the compute and scheduling overheads, we built a simulated FaaS on Tokio [97], the async runtime used in common high performance Rust web services that leverage Wasmtime. We used Wasmtime's epoch_interruption preemption mechanism to preempt workloads at a frequency (epoch) of 1 millisecond, similar to other research [36] and production [27] systems.

Our simulation framework spins up `N` Wasm instances at each epoch (to simulate handling `N` incoming http requests), where `N` is configurable. The instance runs an associated workload involving IO and returns the result from the workload. To simulate the cost of IO, we introduce a delay at the beginning of workloads that we describe below. During this delay, Tokio is free to schedule other Wasm instances, however Wasmtime will not relinquish the memory of any Wasm instance until its workload is complete. The value of the delay is drawn from a Poisson distribution at 5ms; to model typical network request patterns seen in servers [88, 92].

Using this framework, we sought to measure how efficiently ColorGuard and multiprocess scaling strategies handled the same load on one core (all processes were pinned to one core). For our multiprocess strategy, we increase the number of running processes so we can handle the same number of incoming requests as ColorGuard, i.e., we run 15 processes. We then measure the overhead of handling the same number of concurrent requests with each strategy.

Analysis. Figure 6 shows the percent difference in throughput of ColorGuard relative to multiprocess scaling for our three workloads. We see that as the number of processes required to scale increases, ColorGuard provides proportionally greater throughput, offering up to ≈ 29% more throughput vs. multiple processes with a standard deviation of under 3%. As all our workloads are I/O bound, they were able to scale up to 256K concurrent instances. As our simulation keeps I/O latencies uniform, our scaling results are relatively similar across workloads.

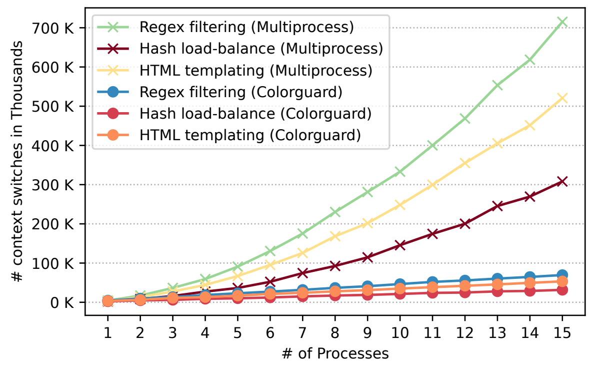
To explain the difference in throughput, we looked at two performance metrics: number of context switches and number of dTLB misses. Figure 7a shows the number of context switches when using ColorGuard vs. process scaling. For ColorGuard, the rate of context switching stays consistent, while the rate of context switches for process scaling increases as we increase the number of processes. This makes sense: as we use more processes they compete for CPU time which causes more context switches and reduces throughput. Figure 7b show the number of dTLB misses when using ColorGuard vs. process scaling. Similar to the case with context switching, as we add more processes, resource contention increases, leading to more dTLB misses and lower throughput.

7 ColorGuard on ARM

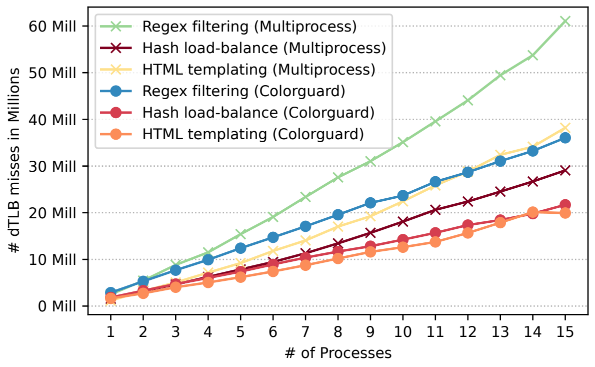
At present, MPK on Intel and AMD is the only widely available hardware support for page coloring, while ARMv9 plans to offer similar support through its permission overlay extension (POE) [9]. We were unable to evaluate ColorGuard using POE, however, as current hardware and emulators do not support POE yet. We do expect that ColorGuard will work with POE with minor changes, as it is exposed through the same Linux system call interface as MPK [40, 68].

ARMv9 also offers a granular coloring of 16-byte memory chunks via its memory tagging extensions (MTE) [7] (memory tagging support is being considered for RISC-V [87]). To explore the viability of ColorGuard-MTE, we prototyped it on a Google Pixel Pro 8 phone, one of the few devices that currently supports MTE. Notably, we found that the available system call support for MTE imposes significant performance penalties — penalties that can be easily avoided.

MTE and ColorGuard. MTE allows applications to tag (color) 16-byte "granules" of memory. These tags are stored in dedicated memory, and can only be updated through MTE-specific instructions. Once memory is tagged, every pointer



(a) Context Switches: Multiprocess Scaling vs. ColorGuard: *As the number of processes required to scale increases, ColorGuard context switches at a constant rate, while the rate of context switching for process scaling increases with each additional process.*



(b) dTLB misses: Multiprocess Scaling vs. ColorGuard: *As the number of processes increases, resource contention (and therefore dTLB misses) increases faster for process scaling than ColorGuard.*

accessing tagged memory must have its' top bits — bits 63 to 60 — set to the same tag as the memory being accessed; if these tags don't match, the processor will trap.

At a high level, ColorGuard-MTE is similar to ColorGuard-MPK: each linear memory is tagged with one color to create a striping pattern (Figure 2); the top bits of a pointer accessing a linear memory are set by the Wasm compiler to ensure the correct tag. By definition, Wasm compilers don't allow programs to use instructions that could modify tags, which guarantees safety of ColorGuard-MTE. In our prototype, we observed two significant sources of performance penalties, and also identified straightforward fixes.

Observation 1: System calls are required for efficient bulk memory tagging. Our Wasm allocator must stripe all linear memories with the appropriate colors during initialization. Unfortunately, doing this with user-level instructions is very slow as MTE allows us to change at most two granules (32 bytes) at a time [7]. MTE's bulk tagging instructions are restricted to kernel code. We tested an example initializing forty 64K linear memories. Without MTE this takes 79 μs per instance, with MTE this takes 2,182 μs. OS support for accessing bulk tagging instructions could potentially mitigate this, and would be a next option to evaluate.

Observation 2: System calls to recycle memory discard tags. Wasm engines such as Wasmtime deallocate a Wasm instance's memory using the `madvise(MADV_DONTNEED)` syscall after it finishes executing. This zeros the memory, but does not remove the mapping, allowing Wasmtime to reuse this mapping for a new instance. Notably with MPK, these per-page colors remain unchanged, meaning the allocator doesn't need to re-stripe memory. In contrast, with MTE, this syscall automatically discards any MTE tags. This is doubly painful: first, the allocator must retag memory for each instance (a slow process per Observation 1); next, this also slows deallocation as tags are cleared — deallocating the forty Wasm instances that we setup earlier goes from 29 μs per instance without MTE, to 377 μs per instance with MTE. Other efforts using MTE for memory safety have also noted that avoiding tag discarding is important for performance [39]. Adding a flag to `madvise` that leaves tags invariant, similar to MPK, would alleviate this cost.

8 Related Work

Wahbe's original system [101], and subsequent work [34, 43, 70, 86, 93, 111] report performance overheads from 20% and 30% or more for SFI. Consequently, reducing these overheads through static analysis [84, 112], different guard page schemes [86, 101, 110] and more efficient control flow integrity [70] have been explored. To the best of our knowledge, Segue offers the largest reduction of SFI overheads on x86-64 in the last two decades.

Segmentation and SFI. x86-64 segmentation can be used to efficiently address separate memory regions. While thread-local-storage (TLS) [50, 58] is the most prominent example, security frameworks [54, 56] have also used this to efficiently address runtime metadata stored by these frameworks in a shadow memory. With Segue, we observe that segment-based addressing is particularly beneficial for SFI, as every heap access can be optimized.

As noted (§3.1), x86-32's segmentation support was used in multiple production [110] and research [34, 60] SFI systems. With Segue, we see that even the small vestiges of segmentation in x86-64 can still benefit SFI systems such as Wasm [43], NaCl [86], LFI [109], etc.

MPK-based isolation. Various research systems have explored using MPK for general in-process isolation [10, 45, 59, 82, 98, 99]. While performant, these systems have trouble scaling because MPK only supports 16 keys (i.e., 16 concurrent sandboxes). Scaling beyond this requires falling back to prohibitively expensive approaches such as page permissions [76] or virtualization [42]. Separately, MPK has also been used by security frameworks that rely on creating just a single isolated domain [54]. In contrast, ColorGuard is the first system to illustrate how MPK can *improve* scaling, by combining MPK with classic SFI techniques.

Multiple works [19, 82, 99] have explored the security challenges MPK-based isolation systems face in restricting unsafe instructions and system calls. However, these are not an issue for Wasm sandboxes, as explained in §3.2.

Reducing register pressure in SFI compilers. Wahbe et al.'s SFI [101] on MIPS and Alpha machines reserved four and five general-purpose registers (GPRs) respectively (out of the available 32), which alone resulted in overheads of up to 7%. Subsequent SFI schemes [70, 86, 93], reduced this to one reserved GPR on x86-64. However, since x86-64 only has 16 GPRs, this still induces register pressure [74]. Segue takes the final step on x86 by eliminating reserved GPRs. While Intel's APX [107] proposes expanding the total GPRs on future CPUs [107], Segue benefits SFI on existing CPUs. That said, a majority of Segue's benefits come from reducing instruction count, thus even with APX, Segue will still be an important optimization.

Reducing guard regions in SFI. Wahbe et al. [101] used guard regions to protect the stack, and observed that heap guard regions could be used as a memory-performance tradeoff — increase the number of guard regions to elide more runtime SFI checks and achieve better performance. NaCl's x86-64 implementation [86] adopted this approach and used 40GB guard regions to optimize performance; but this limited scaling to a maximum of 2,979 sandboxes before exhausting the address space. Wasm [43], in contrast, adopted a 4GB guard region, allowing a maximum of 16,384 sandboxes. As noted in §5, the Wasmtime [16] compiler optimized this further; it reduced guard regions by a factor of 2 by combining 2GB guard regions, with either signed address calculation or select bounds checks. ColorGuard takes this still further, reducing this guard page usage by a factor of 15 by leveraging MPK. This approach can be employed in any SFI scheme that relies on guard regions, such as Wasm [43] or NaCl [86].

Other approaches to in-process isolation. In-process isolation systems based on various hardware features including Intel Memory Protection Extensions (MPX) [61], CET [108], SMAP [102] x86 protection rings [63], virtualization [11, 38, 45], and ARM's memory domains [113] have been explored. However, such approaches come with constraints that have limited their impact on SFI. MPX's overheads are comparable to software implementations [61]; CET and SMAP is restricted to 1 isolated domain [108]; SMAP, ring, virtualization, and Memory Domains incur expensive context switches due to ring/privilege switches [9, 50] and require in-kernel support. In contrast, Segue and ColorGuard offer practical benefits to existing SFI systems and are deployed today. A detailed history of in-process isolation is covered by Tan [93].

Larger virtual address spaces. Newer chips may support address spaces larger than 48-bits which can be used to improve the scalability of SFI. However, the use of larger address spaces come with extra challenges that nevertheless make ColorGuard a preferable solution in many settings.

For instance, some high-end x86-64 server-class processors can be configured to use 57-bit address spaces. However, this requires moving from 4-level to 5-level page tables [47], increasing the cost of a TLB miss by 25%; TLB misses are already significant source of overhead in high-performance Wasm-FaaS platforms as they constantly map and unmap Wasm-heaps. RISC-V's optional sv57 extension [80] similarly offers 57-bit addresses with similar limitations.

On ARM, an optional extension called Large Virtual Addressing (LVA [9]) offers 52-bits virtual addresses. However, LVA increases the minimum page size to 64k in order to support this with 4-level page tables. Thus, it increases memory waste due to internal fragmentation, and also impacts compatibility. For example, `mmap(MAP_FIXED)` is not guaranteed to work if an allocation is not page aligned.

Custom CPU extensions. Multiple processor extensions have been proposed to support in-process isolation [12, 74, 83, 105]. Some offer the same benefits as our optimizations — for instance, HFI [74] eliminates the base-addition in SFI tools, and multiple proposals [12, 74, 83, 105] eliminate guard regions. However, Segue and ColorGuard allow SFI toolchains to offer these benefits on existing CPUs.

9 Conclusion

In the last few years, Wasm has made SFI a critical technology for the internet. However, SFI's performance and scaling constraints impact existing users, and create a barrier to even greater adoption.

Segue uses x86-64 segmentation to bring the cost of SFI instrumented memory operations down to a single instruction (the same as non-Wasm, non-SFI code), often eliminating well over half of SFI's overhead.

ColorGuard uses memory coloring hardware (e.g, MPK) to allow Wasm instances to be packed up to 15× more densely, enabling large scale concurrency with the performance and simplicity benefits of running in a single address space.

Acknowledgment

Thanks to Chris Fallin for his insightful discussions and feedback on this work. This work was supported by the Sloan Foundation, National Science Foundation under Grant Nos. 2327337, 2327336, 2154964, 2155235, 2048262, and 2146755; and by gifts from Intel, Cisco, Mozilla and the Google V8 team. And finally, thanks to our families, without whose support this work would not be possible.

References

[1] Akamai. Serverless computing with akamai edgeworkers. https://www.akamai.com/products/serverless-computing-edgeworkers. Accessed: 2024-01-01.

[2] AMD. AMD64 Architecture Programmer's Manual Volume 3: General-Purpose and System Instructions. https://www.amd.com/content/dam/amd/en/documents/processor-tech-docs/programmer-references/24594.pdf, 2024.

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

[4] Andreas Rossberg (Ed.). Bulk memory operations proposal for webassembly. https://github.com/WebAssembly/bulk-memory-operations, 2021.

[5] Andreas Rossberg (Ed.). Multi memory proposal for webassembly. https://github.com/WebAssembly/multi-memory, 2022.

[6] Anonymous. Segue pr in WAMR. Anonymized for double-blind reviewing, 2023.

[7] ARM. Armv8.5-a memory tagging extension. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf.

[8] ARM. Permission Overlays. https://developer.arm.com/documentation/102376/0200/Permission-indirection-and-permission-overlay-extensions/Permission-overlays.

[9] Arm. ARM architecture reference manual for A-profile architecture. https://developer.arm.com/documentation/ddi0487/latest/, 2024.

[10] Inyoung Bang, Martin Kayondo, Hyungon Moon, and Yunheung Paek. {TRust}: A compilation framework for in-process isolation to protect safe rust against untrusted code. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 6947–6964, 2023.

[11] Adam Belay, Andrea Bittau, Ali José Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. Dune: Safe user-level access to privileged CPU features. In *OSDI*. USENIX, 2012.

[12] Atri Bhattacharyya, Florian Hofhammer, Yuanlong Li, Siddharth Gupta, Andres Sanchez, Babak Falsafi, and Mathias Payer. Securecells: A secure compartmentalized architecture. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2921–2939. IEEE, 2023.

[13] James Bucek, Klaus-Dieter Lange, and Jóakim v. Kistowski. Spec cpu2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, page 41–42, New York, NY, USA, 2018. Association for Computing Machinery.

[14] Sandbox libexpat using rlbox. https://bugzilla.mozilla.org/show_bug.cgi?id=1688452#c37, November 2021.

[15] Bytecode Alliance. Sightglass: a benchmark suite and tool to compare different implementations of the same primitives. https://github.com/bytecodealliance/sightglass, 2019.

[16] Bytecode Alliance. Wasmtime. https://wasmtime.dev, 2021.

[17] ByteCode Alliance. Wasmtime: wasmtime/benches/call.rs. https://github.com/bytecodealliance/wasmtime/blob/main/benches/call.rs, 2024.

[18] Varnish HTTP Cache. Varnish modules. https://developer.fastly.com/solutions/examples/. Accessed: 2024-01-01.

[19] R Joseph Connor, Tyler McDaniel, Jared M Smith, and Max Schuchard. {PKU} pitfalls: Attacks on {PKU-based} memory isolation systems. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1409–1426, 2020.

[20] Russ Cox, Robert Griesemer, Rob Pike, Ian Lance Taylor, and Ken Thompson. The go programming language and environment. *Communications of the ACM*, 65(5):70–78, 2022.

[21] John H. Crawford and Patrick P. Gelsinger. *Programming the 80386*. Sybex Books, 1987.

[22] Alex Crichton. Cve-2022-31104: Miscompilation of i8x16.swizzle and select with v128 inputs. https://www.cve.org/CVERecord?id=CVE-2022-31104, June 2022.

[23] Alex Crichton. Cve-2023-26489: Guest-controlled out-of-bounds read/write on x86_64. https://www.cve.org/CVERecord?id=CVE-2023-26489, June 2023.

[24] WebAssembly Memory64 Discussions. Memory64: bounds-checking strategies. https://github.com/WebAssembly/memory64/issues/3#issuecomment-700841972, 2020.

[25] Duncan Uszkay. How Shopify uses WebAssembly outside of the browser. https://shopify.engineering/shopify-webassembly, 2020.

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

[27] Engineers at large (anonymized for submission) FaaS/CDN provider. private/direct communication.

[28] Engineers at Mozilla Firefox. private/direct communication.

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

[30] Chris Fallin. Cve-2021-32629: Memory access due to code generation flaw in cranelift module. https://www.cve.org/CVERecord?id=CVE-2021-32629, May 2021.

[31] Chris Fallin. Wasmtime 1.0: A look at performance. https://bytecodealliance.org/articles/wasmtime-10-performance, September 2022.

[32] Fastly. Fastly: Code examples. https://developer.fastly.com/solutions/examples/. Accessed: 2024-01-01.

[33] Adam Foltzer. The lifecycle and performance of a lucet instance. https://www.fastly.com/blog/lucet-performance-and-lifecycle, 2019. Accessed: 2022-08-10.

[34] Bryan Ford and Russ Cox. Vx32: Lightweight user-level sandboxing on the x86. In *Proceedings of USENIX ATC 2008*. USENIX, 2008.

[35] Nathan Froyd. Securing Firefox with WebAssembly. https://hacks.mozilla.org/2020/02/securing-firefox-with-webassembly/, 2020.

[36] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. Sledge: a serverless-first, lightweight wasm runtime for the edge. In *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*. ACM, 2020.

[37] Gcc. Named address spaces. https://gcc.gnu.org/onlinedocs/gcc-9.1.0/gcc/Named-Address-Spaces.html.

[38] Nuwan Goonasekera, William Caelli, and Colin Fidge. LibVM: an architecture for shared library sandboxing. *Software: Practice and Experience*, 45(12), 2015.

[39] Floris Gorter, Taddeus Kroes, Herbert Bos, and Cristiano Giuffrida. Sticky Tags: Efficient and Deterministic Spatial Memory Error Mitigation using Persistent Memory Tags. In *S&P*, May 2024.

[40] Joey Gouly. [PATCH] Permission Overlay Extension . https://patchwork.kernel.org/project/linux-fsdevel/cover/20231124163510.1835740-1-joey.gouly@arm.com/, November 2023.

[41] Graphite - A free and open rendering engine for complex scripts. http://scripts.sil.org/RenderingGraphite, 2012.

[42] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. Epk: Scalable and efficient memory protection keys. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2022.

[43] Andreas Haas, Andreas Rossberg, Derek L Schuff, Ben L Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. Bringing the web up to speed with WebAssembly. In *PLDI*. ACM, 2017.

[44] L. Hansen. Mark the jump_table_entry instruction as loading. https://github.com/bytecodealliance/cranelift/pull/805, 2019.

[45] Mohammad Hedayati, Spyridoula Gravani, Ethan Johnson, John Criswell, Michael L Scott, Kai Shen, and Mike Marty. Hodor: Intra-process isolation for high-throughput data plane libraries. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association, 2019.

[46] Bobby Holley. WebAssembly and back again: Fine-grained sandboxing in Firefox 95. https://hacks.mozilla.org/2021/12/webassembly-

and-back-again-fine-grained-sandboxing-in-firefox-95/, November 2021.

[47] Intel. 5-level paging and 5-level ept white paper. https://www.intel.com/content/www/us/en/content-details/671442/5-level-paging-and-5-level-ept-white-paper.html, May 2017.

[48] Intel. WebAssembly Micro Runtime. https://github.com/bytecodealliance/wasm-micro-runtime, 2020.

[49] Intel. Envisioning a simplified intel architecture. https://www.intel.com/content/www/us/en/developer/articles/technical/envisioning-future-simplified-architecture.html, November 2023.

[50] Intel® 64 and IA-32 architectures software developer's manual. https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html, 2023.

[51] Intel. Wamr release 1.2.3. https://github.com/bytecodealliance/wasm-micro-runtime/releases/tag/WAMR-1.2.3, 2023.

[52] Intel. WAMR vectorization optimizations. https://github.com/bytecodealliance/wasm-micro-runtime/blob/b3f728ceb36f9c72047a934436ef41699643ab99/core/iwasm/compilation/aot_llvm_extra.cpp#L330, 2024.

[53] formatted by felix coultier Intel. x86 and amd64 instruction reference. https://www.felixcloutier.com/x86/wrpkru, October 2024.

[54] Mohannad Ismail, Jinwoo Yom, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. Vip: Safeguard value invariant property for thwarting critical memory corruption attacks. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1612–1626, 2021.

[55] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *ATC*. USENIX, 2019.

[56] Christopher Jelesnianski, Mohannad Ismail, Yeongjin Jang, Dan Williams, and Changwoo Min. Protect the system call, protect (most of) the world with bastion. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*, pages 528–541, 2023.

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

[58] kernel.org. The linux kernel documentation. https://www.kernel.org/doc/html/next/x86/x86_64/fsgs.html, June 2024.

[59] Paul Kirth, Mitchel Dickerson, Stephen Crane, Per Larsen, Adrian Dabrowski, David Gens, Yeoul Na, Stijn Volckaert, and Michael Franz. Pkru-safe: Automatically locking down the heap between safe and unsafe languages. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 132–148, 2022.

[60] Matthew Kolosick, Shravan Narayan, Conrad Watt, Michael LeMay, Deepak Garg, Ranjit Jhala, and Deian Stefan. Isolation without taxation: Near zero cost transitions for sfi. In *Proceedings of the ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*. ACM, January 2022.

[61] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. No need to hide: Protecting safe regions on commodity hardware. In *EuroSys*. ACM, 2017.

[62] Butler W Lampson. Protection. *ACM SIGOPS Operating Systems Review*, 8(1):18–24, 1974.

[63] Hojoon Lee, Chihyun Song, and Brent Byunghoon Kang. Lord of the x86 rings: A portable user mode privilege separation architecture on x86. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1441–1454, 2018.

[64] Nico Lehmann, Adam T Geller, Niki Vazou, and Ranjit Jhala. Flux: Liquid types for rust. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1533–1557, 2023.

[65] Expat XML parser. https://libexpat.github.io/.

[66] Louis-Noël Pouchet. Polybench/c: the polyhedral benchmark suite. https://web.archive.org/web/20231102034252/http://web.cse.ohio-state.edu/~pouchet.2/software/polybench/.

[67] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My VM is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles (OSDI)*, 2017.

[68] Catalin Marinas. Linux 6.5 ARM updates. https://lore.kernel.org/lkml/20230626174435.1791242-1-catalin.marinas@arm.com/, June 2023.

[69] Stephen McCamant and Greg Morrisett. Efficient, verifiable binary sandboxing for a CISC architecture. *CSAIL Tech report*, 2005.

[70] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *Security*. USENIX, 2006.

[71] Tyler McMullen. Lucet: A compiler and runtime for high-concurrency low-latency sandboxing. In *PriSC*, 2020.

[72] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. Retrofitting fine grain isolation in the Firefox renderer. In *SEC*. USENIX, 2020.

[73] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, and Deian Stefan. Swivel: Hardening {WebAssembly} against spectre. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 1433–1450, 2021.

[74] Shravan Narayan, Tal Garfinkel, Mohammadkazem Taram, Joey Rudek, Daniel Moghimi, Evan Johnson, Chris Fallin, Anjo Vahldiek-Oberwagner, Michael LeMay, Ravi Sahita, et al. 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*, pages 266–281, 2023.

[75] Robert M. Norton. Hardware support for compartmentalisation. Technical Report UCAM-CL-TR-887, University of Cambridge, Computer Laboratory, May 2016.

[76] Soyeon Park, Sangho Lee, Wen Xu, Hyungon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (intel mpk). In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 241–254, 2019.

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

[78] Pat Hickey. How Fastly and the developer community are investing in the WebAssembly ecosystem. https://www.fastly.com/blog/how-fastly-and-developer-community-invest-in-webassembly-ecosystem, 2020.

[79] Pengyuan Bian. Istio and Envoy WebAssembly extensibility, one year on. https://istio.io/latest/blog/2021/wasm-progress/, 2021.

[80] RISC-V. Risc-v gets sv57-based virtual memory. https://riscv.org/news/2022/03/risc-v-gets-sv57-based-virtual-memory-other-improvements-for-linux-5-18-michael-larabel-phoronix/, March 2022.

[81] Henrik Rydgard. Windows (Fastcall) calling convention: Callee-saved XMM (FP) registers are not actually saved. https://github.com/bytecodealliance/wasmtime/issues/1177, 2020.

[82] David Schrammel, Samuel Weiser, Richard Sadek, and Stefan Mangard. Jenny: Securing syscalls for {PKU-based} memory isolation systems. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 936–952, 2022.

[83] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys – efficient in-process isolation for risc-v and x86. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1677–1694. USENIX Association, August 2020.

[84] Mark Seaborn. Sandboxing libraries in Chrome using SFI: zlib proof-of-concept. https://docs.google.com/presentation/d/1RD3bxsBfTZOIfrlq7HzGMsygPHgb61A1eTdelIYOurs/, 2013.

[85] Korakit Seemakhupt, Brent E Stephens, Samira Khan, Sihang Liu, Hassan Wassel, Soheil Hassas Yeganeh, Alex C Snoeren, Arvind Krishnamurthy, David E Culler, and Henry M Levy. A cloud-scale characterization of remote procedure calls. In *Proceedings of the 29th Symposium on Operating Systems Principles*, pages 498–514, 2023.

[86] David Sehr, Robert Muth, Cliff Biffle, Victor Khimenko, Egor Pasko, Karl Schimpf, Bennet Yee, and Brad Chen. Adapting software fault isolation to contemporary CPU architectures. In *Security*. USENIX, 2010.

[87] Rafael Sene. Risc-v memory tagging task group. https://github.com/riscv-admin/riscv-memory-tagging, May 2024.

[88] Mohammad Shahrad, Rodrigo Fonseca, Inigo Goiri, Gohar Chaudhry, Paul Batum, Jason Cooke, Eduardo Laureano, Colby Tresness, Mark Russinovich, and Ricardo Bianchini. Serverless in the wild: Characterizing and optimizing the serverless workload at a large cloud provider. In *2020 USENIX annual technical conference (USENIX ATC 20)*, pages 205–218, 2020.

[89] Igor Sheludko and Santiago Aboy Solanes. Pointer compression in V8. https://v8.dev/blog/pointer-compression, 2020.

[90] Simon Shillaker and Peter Pietzuch. FAASM: Lightweight isolation for efficient stateful serverless computing. In *ATC*. USENIX, 2020.

[91] SingleStore. Docs: Code engine - powered by wasm. https://docs.singlestore.com/cloud/reference/code-engine-powered-by-wasm/.

[92] Jovan Stojkovic, Chunao Liu, Muhammad Shahbaz, and Josep Torrellas. μmanycore: A cloud-native cpu for tail at scale. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, pages 1–15, 2023.

[93] Gang Tan. Principles and implementation techniques of software-based fault isolation. *Foundations and Trends in Privacy and Security*, 1(3), 2017.

[94] The WebAssembly Binary Toolkit. wasm2c. https://github.com/WebAssembly/wabt/tree/master/wasm2c, 2018.

[95] Thomas Nattestad. WebAssembly brings Google Earth to more browsers. https://blog.chromium.org/2019/06/webassembly-brings-google-earth-to-more.html, 2019.

[96] Nabeel Al-Shamma Thomas Nattestad. Photoshop's journey to the web. https://web.dev/ps-on-the-web/, 2022.

[97] Tokio. An asynchronous rust runtime. https://tokio.rs/. Accessed: 2024-01-01.

[98] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys (MPK). In *Security*. USENIX, 2019.

[99] Alexios Voulimeneas, Jonas Vinck, Ruben Mechelinck, and Stijn Volckaert. You shall not (by) pass! practical, secure, and fast pku-based sandboxing. In *Proceedings of the Seventeenth European Conference on Computer Systems*, pages 266–282, 2022.

[100] Luke Wagner. Component model design and specification. https://github.com/WebAssembly/component-model.

[101] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *SOSP*. ACM, 1993.

[102] Zhe Wang, Chenggang Wu, Mengyao Xie, Yinqian Zhang, Kangjie Lu, Xiaofeng Zhang, Yuanming Lai, Yan Kang, and Min Yang. Seimi: Efficient and secure smap-enabled intra-process memory isolation. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 592–607. IEEE, 2020.

[103] WebAssembly system interface. https://wasi.dev. Accessed: 2024-01-01.

[104] Wasmtime. Security advisories. https://github.com/bytecodealliance/wasmtime/security/advisories, February 2024.

[105] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. Cheri: A hybrid capability-system architecture for scalable software compartmentalization. In *2015 IEEE Symposium on Security and Privacy*, pages 20–37, 2015.

[106] Reinhold P Weicker. Dhrystone: a synthetic systems programming benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

[107] Sebastian Winkel and Jason Agron. Introducing intel advanced performance extensions (intel apx). https://www.intel.com/content/www/us/en/developer/articles/technical/advanced-performance-extensions-apx.html, August 2023.

[108] Mengyao Xie, Chenggang Wu, Yinqian Zhang, Jiali Xu, Yuanming Lai, Yan Kang, Wei Wang, and Zhe Wang. Cetis: Retrofitting intel cet for generic and efficient intra-process memory isolation. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 2989–3002, 2022.

[109] Zachary Yedidia. Lightweight fault isolation: Practical, efficient, and secure software sandboxing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS '24, page 649–665, New York, NY, USA, 2024. Association for Computing Machinery.

[110] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *S&P*. IEEE, 2009.

[111] Alon Zakai. Wasmboxc: Simple, easy, and fast vm-less sandboxing. https://kripken.github.io/blog/wasm/2020/07/27/wasmboxc.html, 2020.

[112] Bin Zeng, Gang Tan, and Greg Morrisett. Combining control-flow integrity and static analysis for efficient and validated data sandboxing. In *CCS*, 2011.

[113] Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. Armlock: Hardware-based fault isolation for arm. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. ACM, 2014.