Swivel: Hardening WebAssembly against Spectre

Shravan Narayan[†] Craig Disselkoen[†] Daniel Moghimi[¶†]
Sunjay Cauligi[†] Evan Johnson[†] Zhao Gang[†]
Anjo Vahldiek-Oberwagner[⋆] Ravi Sahita[*] Hovav Shacham[‡] Dean Tullsen[†] Deian Stefan[†]
[†]UC San Diego [¶]Worcester Polytechnic Institute [⋆]Intel Labs [*]Intel [‡]UT Austin

Abstract

We describe Swivel, a new compiler framework for hardening WebAssembly (Wasm) against Spectre attacks. Outside the browser, Wasm has become a popular lightweight, in-process sandbox and is, for example, used in production to isolate different clients on edge clouds and function-as-a-service platforms. Unfortunately, Spectre attacks can bypass Wasm's isolation guarantees. Swivel hardens Wasm against this class of attacks by ensuring that potentially malicious code can neither use Spectre attacks to break out of the Wasm sandbox nor coerce victim code—another Wasm client or the embedding process—to leak secret data.

We describe two Swivel designs, a software-only approach that can be used on existing CPUs, and a hardware-assisted approach that uses extension available in Intel® 11th generation CPUs. For both, we evaluate a randomized approach that mitigates Spectre and a deterministic approach that eliminates Spectre altogether. Our randomized implementations impose under 10.3% overhead on the Wasm-compatible subset of SPEC 2006, while our deterministic implementations impose overheads between 3.3% and 240.2%. Though high on some benchmarks, Swivel's overhead is still between 9× and 36.3× smaller than existing defenses that rely on pipeline fences.

1 Introduction

WebAssembly (Wasm) is a portable bytecode originally designed to safely run native code (e.g., C/C++ and Rust) in the browser [27]. Since its initial design, though, Wasm has been increasingly used to sandbox untrusted code outside the browser. For example, Fastly and Cloudflare use Wasm to sandbox client applications running on their edge clouds—where multiple client applications run within a single process [30, 85]. Mozilla uses Wasm to sandbox third-party C/C++ libraries in Firefox [21, 65]. Yet others use Wasm to isolate untrusted code in serverless computing [28], IoT applications [10], games [62], trusted execution environments [17], and even OS kernels [79].

In this paper, we focus on hardening Wasm against Spectre attacks—the class of transient execution attacks which exploit control flow predictors [49]. Transient execution attacks which exploit features within the memory subsystem (e.g., Meltdown [57], MDS [11, 72, 84], and Load Value Injection [83]) are limited in scope and have already been fixed in recent microarchitectures [41] (see Section 6.2). In contrast, Spectre can allow attackers to bypass Wasm's isolation boundary on almost all superscalar CPUs [3, 4, 35]—and, unfortunately, current mitigations for Spectre cannot be implemented entirely in hardware [5, 13, 43, 51, 59, 76, 81, 93].

On multi-tenant serverless, edge-cloud, and function as a service (FaaS) platforms, where Wasm is used as *the* way to isolate mutually disturbing tenants, this is particulary concerning:[1] A malicious tenant can use Spectre to break out of the sandbox and read another tenant's secrets in two steps (§5.4). First, they *mistrain* different components of the underlying control flow prediction—the conditional branch predictor (CBP), branch target buffer (BTB), or return stack buffer (RSB)—to speculatively execute code that accesses data outside the sandbox boundary. Then, they *reconstruct* the secret data from the underlying microarchitectural state (typically the cache) using a side channel (cache timing).

One way to mitigate such Spectre-based *sandbox breakout attacks* is to partition mutually distrusting code into separate processes. By placing untrusted code in a separate process we can ensure that the attacker cannot access secrets. Chrome and Firefox, for example, do this by partitioning different *sites* into separate processes [25, 64, 68]. On a FaaS platform, we could similarly place tenants in separate processes.

Unfortunately, this would still leave tenants vulnerable to cross-process *sandbox poisoning attacks* [12, 36, 50]. Specifically, attackers can poison hardware predictors to coerce a victim sandbox to speculatively execute gadgets that access secret data—from their own memory region—and leak it via the cache (e.g., by branching on the secret). Moreover, using process isolation would sacrifice Wasm's scalability (running many sandboxes within a process) and performance (cheap startup times and context switching) [28, 30, 65, 85].

The other popular approach, removing speculation within the sandbox, is also unsatisfactory. For example, using pipeline fences to restrict Wasm code to sequential execution imposes a 1.8×–7.3× slowdown on SPEC 2006 (§5). Conservatively inserting pipeline fences before every dynamic load—an approach inspired by the mitigation available in Microsoft's Visual Studio compiler [61]—is even worse: it incurs a 7.3×–19.6× overhead on SPEC (§5).

[1]Though our techniques are general, for simplicity we henceforth focus on Wasm as used on FaaS platforms.

In this paper, we take a compiler-based approach to hardening Wasm against Spectre, without resorting to process isolation or the use of fences. Our framework, Swivel, addresses not only sandbox breakout and sandbox poisoning attacks, but also *host poisoning attacks*, i.e., Spectre attacks that coerce the process hosting the Wasm sandboxes into leaking sensitive data. That is, Swivel ensures that a malicious Wasm tenant cannot speculatively access data outside their sandbox nor coerce another tenant or the host to divulge secrets of other sandboxes via poisoning. We develop Swivel via three contributions:

1. Software-only Spectre hardening (§3.2) Our first contribution, Swivel-SFI, is a software-only approach to hardening Wasm against Spectre. Swivel-SFI eliminates sandbox breakout attacks by compiling Wasm code to *linear blocks (LBs)*. Linear blocks are straight-line x86 code blocks that satisfy two invariants: (1) all transfers of control, including function calls, are at the block boundary—to (and from) other linear blocks; and (2) all memory accesses within a linear block are masked to the sandbox memory. These invariants are necessary to ensure that the speculative control and data flow of the Wasm code is restricted to the sandbox boundary. They are not sufficient though: Swivel-SFI must also be tolerant of possible RSB underflow. We address this by (1) not emitting `ret` instructions and therefore completely bypassing the RSB and (2) using a *separate stack* for return addresses.

To address poisoning attacks, Swivel-SFI must still account for a poisoned BTB or CBP. Since these attacks are more sophisticated, we evaluate two different ways of addressing them, and allow tenants to choose between them according to their trust model. The first approach uses address space layout randomization (ASLR) to randomize the placement of each Wasm sandbox and flushes the BTB on each sandbox boundary crossing. This does not eliminate poisoning attacks; it only raises the bar of Wasm isolation to that of process isolation. Alternately, tenants can opt to eliminate these attacks altogether; to this end, our deterministic Swivel-SFI rewrites conditional branches to indirect jumps—thereby completely bypassing the CBP (which cannot be directly flushed) and relying solely on the BTB (which can).

2. Hardware-assisted Spectre hardening (§3.3) Our second contribution, Swivel-CET, restores the use of all predictors, including the RSB and CBP, and partially obviates the need for BTB flushing. It does this by sacrificing backwards compatibility and using new hardware security extensions: Intel's Control-flow Enforcement Technology (CET) [39] and Memory Protection Keys (MPK) [39].

Like Swivel-SFI, Swivel-CET relies on linear blocks to address sandbox breakout attacks. But Swivel-CET does not avoid `ret` instructions. Instead, we use Intel® CET's hardware *shadow stack* to ensure that the RSB cannot be misused to speculatively return to a location that is different from the expected function return site on the stack [39].

To eliminate host poisoning attacks, we use both Intel® CET and Intel® MPK. In particular, we use Intel® MPK to partition the application into two domains—the host and Wasm sandbox(es)—and, on context switch, ensure that each domain can only access its own memory regions. We use Intel® CET forward-edge control-flow integrity to ensure that application code cannot jump, sequentially or speculatively, into arbitrary sandbox code (e.g., due to a poisoned BTB). We do this by inserting `endbranch` instructions in Wasm sandboxes to demarcate valid jump targets, essentially partitioning the BTB into two domains. Our use of Intel's MPK and CET ensures that even if the host code runs with poisoned predictors, it cannot read—and thus leak—sandbox data.

Since Intel® MPK only supports 16 protection regions, we cannot use it to similarly prevent sandbox poisoning attacks: serverless, edge-cloud, and FaaS platforms have thousands of co-located tenants. Hence, to address sandbox poisoning attacks, like for Swivel-SFI, we consider and evaluate two designs. The first (again) uses ASLR to randomize the location of each sandbox and flushes the BTB on sandbox entry; we don't flush on sandbox exit since the host can safely run with a poisoned BTB. The second is deterministic and not only allows using conditional branches but also avoids BTB flushes. It does this using a new technique, *register interlocking*, which tracks the control flow of the Wasm sandbox and turns every misspeculated memory access into an access to an empty guard page. Register interlocking allows a tenant to run with a poisoned BTB or CBP since any potentially poisoned speculative memory accesses will be invalidated.

3. Implementation and evaluation (§4–5) We implement both Swivel-SFI and Swivel-CET by modifying the Lucet compiler's Wasm-to-x86 code generator (Cranelift) and runtime. To evaluate Swivel's security we implement proof of concept breakout and poisoning attacks against stock Lucet (mitigated by Swivel). We do this for all three Spectre variants, i.e., Spectre attacks that abuse the CBP, BTB, and RSB.

We evaluate Swivel's performance against stock Lucet and several fence-insertion techniques on several standard benchmarks. On the Wasm compatible subset of the SPEC 2006 CPU benchmarking suite we find the ASLR variants of Swivel-SFI and Swivel-CET impose little overhead—they are at most 10.3% and 6.1% slower than Lucet, respectively. Our deterministic implementations, which eliminate all three categories of attacks, incur modest overheads: Swivel-SFI and Swivel-CET are respectively 3.3%–86.1% (geomean: 47.3%) and 8.0%–240.2% (geomean: 96.3%) slower than Lucet. These overheads are smaller than the overhead imposed by state-of-the-art fence-based techniques.

Open source and data We make all source and data available under an open source license at: https://swivel.programming.systems.

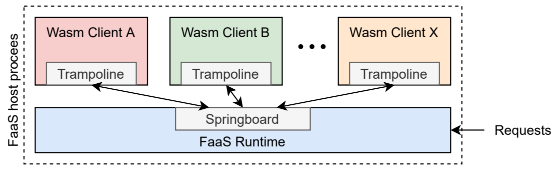


Figure 1: FaaS platform using Wasm to isolate mutually distrusting tenants.

2 A brief overview of Wasm and Spectre

In this section, we give a brief overview of WebAssembly's use in multi-tenant serverless and edge-cloud computing platforms—and more generally function as a service (FaaS) platforms. In particular, we describe how FaaS platforms use Wasm as an intermediate compilation layer for isolating different tenants today. We then briefly review Spectre attacks and describe how today's approach to isolating Wasm code is vulnerable to this class of attacks.

2.1 WebAssembly

Wasm is a low-level 32-bit machine language explicitly designed to embed C/C++ and Rust code in host applications. Wasm programs (which are simply a collection of functions) are (1) deterministic and well-typed, (2) follow a structured control flow discipline, and (3) separate the heap—the *linear memory*—from the well-typed stack and program code. These properties make it easy for compilers like Lucet to sandbox Wasm code and safely embed it within an application [27].

Control flow safety Lucet's code generator, Cranelift, ensures that the control flow of the compiled code is restricted to the sandbox and cannot be bent to bypass bounds checks (e.g., via return-oriented programming). This comes directly from preserving Wasm's semantics during compilation. For example, compiled code preserves Wasm's safe stack [27, 52], ensuring that stack frames (and thus return values on the stack) cannot be clobbered. The compiled code also enforces Wasm's coarse-grained CFI and, for example, matches the type of each indirect call site with the type of the target.

Memory isolation When Lucet creates a Wasm sandbox, it reserves 4GB of virtual memory for the Wasm heap and uses Cranelift to bound all heap loads and stores. To this end, Cranelift (1) explicitly passes a pointer to the base of the heap as the first argument to each function and (2) masks all pointers to be within this 4GB range. Like previous software-based isolation (SFI) systems [88], Cranelift avoids expensive bounds check operations by using guard pages to trap any offsets that may reach beyond the 4GB heap space.

Embedding Wasm An application with embedded Wasm code will typically require context switching between the Wasm code and host—e.g., to read data from a socket. Lucet allows safe control and data flow across the host-sandbox boundary via *springboards* and *trampolines*. Springboards are used to enter Wasm code—they set up the program context for Wasm execution—while trampolines are used to restore the host context and resume execution in the host.

Using Wasm in FaaS platforms WebAssembly FaaS platforms like Fastly's Terrarium allow clents to deploy scalable function-oriented Web and cloud applications written in any langauge (that can be compiled to Wasm). Clients compile their code to Wasm and upload the resulting module to the platform; the platform handles scaling and isolation. As shown in Figure 1, FaaS platforms place thousands of client Wasm modules within a single host process, and distribute these processes across thousands of servers and multiple datacenters. This is the key to scaling—it allows any host process, in any datacenter, to spawn a fresh Wasm sandbox instance and run any client function in response to a user request. By using Wasm as an intermediate layer, FaaS platforms isolate the client for free [6, 30, 60, 65, 66, 88]. Unfortunately, this isolation does not hold in the presence of Spectre attacks.

2.2 Spectre attacks

Spectre attacks exploit hardware predictors to induce mispredictions and speculatively execute instructions—*gadgets*—that would not run sequentially [49]. Spectre attacks are classified by the hardware predictor they exploit [12]. We focus on the three Spectre variants that hijack control flow:

- **Spectre-PHT** Spectre-PHT [49] exploits the *pattern history table* (PHT), which is used as part of the *conditional branch predictor* (CBP) to guess the direction of a conditional branch while the condition is still being evaluated. In a Spectre-PHT attack, the attacker (1) pollutes entries in the PHT so that a branch is mispredicted to the wrong path. The attacker can then use this wrong-path execution to bypass memory isolation guards or control flow integrity.
- **Spectre-BTB** Spectre-BTB [49] exploits the *branch target buffer* (BTB), which is used to predict the target of an indirect jump [94]. In a Spectre-BTB attack, the attacker pollutes entries in the BTB, redirecting speculative control flow to an arbitrary target. Spectre-BTB can thus be used to speculatively execute gadgets that are not in the normal execution path (e.g., to carry out a ROP-style attack).
- **Spectre-RSB** Spectre-RSB [50, 58] exploits the *return stack buffer* (RSB), which memorizes the location of recently executed `call` instructions to predict the targets of `ret` instructions. In a Spectre-RSB attack, the attacker uses chains of `call` or `ret` instructions to over- or underflow the RSB, and redirect speculative control flow in turn.

Spectre can be used *in-place* or *out-of-place* [12]. In an in-place attack, the attacker mistrains the prediction for a victim branch by repeatedly executing the victim branch itself. In an out-of-place attack, the attacker finds a secondary branch that is *congruent* to the victim branch—predictor entries are indexed using a subset of address bits—and uses this secondary branch to mistrain the prediction for the victim branch.

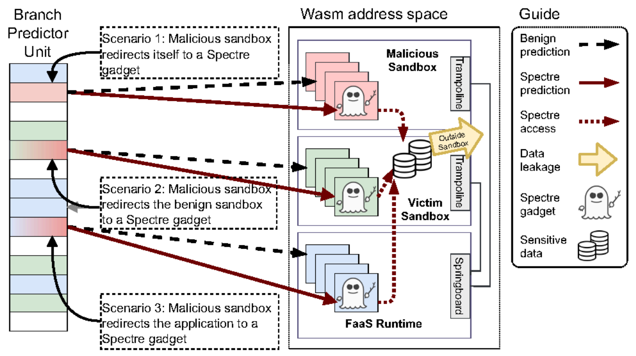


Figure 2: A malicious tenant can fill branch predictors with invalid state (red). In one scenario, the attacker causes its own branches to speculatively execute code that access memory outside of the sandbox. In the second and third scenarios, the attacker uses Spectre to respectively target a victim sandbox or the host runtime to misspeculate and leak secret data.

2.3 Spectre attacks on FaaS platforms

A malicious FaaS platform client who can upload arbitrary Wasm code can force the Wasm compiler to emit native code which is safe during sequential execution, but uses Spectre to bypass Wasm's isolation guarantees during speculative execution. We identify three kinds of attacks (Figure 2):

- **Scenario 1: Sandbox breakout attacks** The attacker bends the speculative control flow of their own module to access data outside the sandbox region. For example, they can use Spectre-PHT to bypass conditional bounds checks when accessing the indirect call table. Alternatively, they can use Spectre-BTB to transfer the control flow into the middle of instructions to execute unsafe code (e.g., code that bypasses Wasm's implicit heap bounds checks).
- **Scenario 2: Sandbox poisoning attacks** The attacker uses an out-of-place Spectre attack to bend the control flow of a victim sandbox and coerce the victim into leaking their own data. Although this attack is considerably more sophisticated, we were still able to implement proof of concept attacks following Canella et al. [12]. Here, the attacker finds a (mispredicted) path in the victim sandbox that leads to the victim leaking data, e.g., through cache state. They then force the victim to mispredict this path by using a congruent branch within their own sandbox.
- **Scenario 3: Host poisoning attacks** Instead of bending the control flow of the victim sandbox, the attacker can use an out-of-place Spectre attack to bend the control flow of the host runtime. This allows the attacker to speculatively access data from the host as well as any other sandbox.

Figure 3 gives an example sandbox breakout gadget. The gadget is in the implementation of the Wasm `call_indirect` instruction, which is used to call functions indexed in a module-level function table. This code first compares the function index `rcx` to the length of the function table (to ensure that `rcx` points to a valid entry). If `rcx` is valid, it then jumps to `index_ok`, loads the function from the corresponding entry in

```
1 mov    rdx,QWORD PTR [fn_table_len]  ; get fn table length
2 cmp    rcx,rdx  ; check that rcx is in-bounds
3 jb     index_ok
4 ud2  ; trap otherwise
5 index_ok:
6 lea    rdx,[fn_table]
7 mov    rcx,QWORD PTR [rdx+rcx*4]
8 call   rcx
```

Figure 3: A simplified snippet of the vulnerable code from our Spectre-PHT breakout attack. This code is safe during sequential execution (it checks the index `rcx` before using it to load a function table entry). But, during speculative execution, control flow may bypass this check and access memory outside the function table bounds.

Attack variant		Swivel-SFI		Swivel-CET	
		ASLR	Det	ASLR	Det
Spectre-PHT	in-place	●	●	●	●
	out-of-place	◐	●	◐	●
Spectre-BTB	in-place	●	●	●	●
	out-of-place	●	●	●	●
Spectre-RSB	in-place	●	●	●	●
	out-of-place	●	●	●	●

Table 1: Effectiveness of Swivel against different Spectre variants. A full circle indicates that Swivel eliminates the attack while a half circle indicates that Swivel only mitigates the attack.

the table, and calls it; otherwise the code traps.

An attacker can mistrain the conditional branch on line 3 and cause it to speculatively jump to `index_ok` even when `rcx` is out-of-bounds. By controlling the contents of `rcx`, the attacker can thus execute arbitrary code locations outside the sandbox. In Section 5.4 we demonstrate several proof of concept attacks, including a breakout attack that uses this gadget. These attacks serve to highlight the importance of hardening Wasm against Spectre.

3 Swivel: Hardening Wasm against Spectre

Swivel extends Lucet—and the underlying Cranelift code generator—to address Spectre attacks on FaaS Wasm platforms. We designed Swivel with several goals in mind. First, *performance*: Swivel minimizes the number of pipeline fences it inserts, allowing Wasm to benefit from speculative execution as much as possible. Second, *automation*: Swivel does not rely on user annotations or source code changes to guide mitigations; we automatically apply mitigations when compiling Wasm. Finally, *modularity*: Swivel offers configurable protection, ranging from probabilistic schemes with high performance to thorough mitigations with strong guarantees. This allows Swivel users to choose the most appropriate mitigations (see Table 1) according to their application domain, security considerations, or particular hardware platform.

In the rest of this section, we describe our attacker model and introduce a core abstraction: *linear blocks*. We then show how linear blocks, together with several other techniques, are used to address both sandbox breakout and poisoning attacks. These techniques span two Swivel designs: Swivel-SFI, a software-only scheme which provides mitigations compatible with existing CPUs; and Swivel-CET, which uses hardware extensions (Intel® CET and Intel® MPK) available in the 11th generation Intel® CPUs.

Attacker model We assume that the attacker is a FaaS platform client who can upload arbitrary Wasm code which the platform will then compile and run alongside other clients using Swivel. The goal of the attacker is to read data sensitive to another (victim) client using Spectre attacks. In the Swivel-CET case, we only focus on exfiltration via the data cache—and thus assume an attacker who can only exploit gadgets that leak via the data cache. We consider transient attacks that exploit the memory subsystem (e.g., Meltdown [57], MDS [11, 72, 84], and LVI [83]) out of scope and discuss this in detail in Section 6.2.

We assume that our Wasm compiler and runtime are correct. We similarly assume the underlying operating system is secure and the latest CPU microcode updates are applied. We assume hyperthreading is disabled for any Swivel scheme except for the deterministic variant of Swivel-CET. Consistent with previous findings [94], we assume BTBs predict the lower 32-bits of target addresses, while the upper 32-bits are inferred from the instruction pointer.

Swivel addresses attackers that intentionally extract information using Spectre. We do not prevent clients from accidentally leaking secrets during sequential execution and, instead, assume they use techniques like constant-time programming to prevent such leaks [14]. For all Swivel schemes except the deterministic variant of Swivel-CET, we assume that a sandbox cannot directly invoke function calls in other sandboxes, i.e., it cannot control the input to another sandbox to perform an in-place poisoning attack. We lastly assume that host secrets can be protected by placing them in a Wasm sandbox, and discuss this further in Section 6.1.

3.1 Linear blocks: local Wasm isolation

To enforce Wasm's isolation sequentially and speculatively, Swivel-SFI and Swivel-CET compile Wasm code to linear blocks (LBs). Linear blocks are straight-line code blocks that do not contain *any* control flow instructions except for their terminators—this is in contrast to traditional basic blocks, which typically do not consider function calls as terminators. This simple distinction is important: It allows us to ensure that *all* control flow transfers—both sequential and speculative—land on linear block boundaries. Then, by ensuring that individual linear blocks are *safe*, we can ensure that whole Wasm programs, when compiled, are confined and cannot violate Wasm's isolation guarantees.

A linear block is safe if, independent of the control flow into the block, Wasm's isolation guarantees are preserved. In particular, we cannot rely on safety checks (e.g., bounds checks for memory accesses) performed across linear blocks since, speculatively, blocks may not always execute in sequential order (e.g., because of Spectre-BTB). When generating native code, Swivel ensures that a linear block is safe by:

Masking memory accesses Since we cannot make any assumptions about the initial contents of registers, Swivel ensures that unconditional heap bounds checks (performed via masking) are performed in the same linear block as the heap memory access itself. We do this by modifying the Cranelift optimization passes which could lift bounds checks (e.g., loop invariant code motion) to ensure that they don't move masks across linear block boundaries. Similarly, since we cannot trust values on the stack, Swivel ensures that any value that is unspilled from the stack and used in a bounds check is masked again. We use this *mask-after-unspill* technique to replace Cranelift's unsafe mask-before-spill approach.

Pinning the heap registers To properly perform bounds checks for heap memory accesses, a Swivel linear block must determine the correct value of the heap base. Unfortunately, as described above, we cannot make any assumptions about the contents of any register or stack slot. Swivel thus reserves one register, which we call the *pinned heap register*, to store the address of the sandbox heap. Furthermore, Swivel prevents any instructions in the sandbox from altering the pinned heap register. This allows each linear block to safely assume that the pinned heap register holds the correct value of the heap base, even when the speculative control flow of the program has gone awry due to misprediction.

Hardening jump tables Wasm requires bounds checks on each access to indirect call tables and switch tables. Swivel ensures that each of these bounds checks is local to the linear block where the access is performed. Moreover, Swivel implements the bounds check using *speculative load hardening* [13], masking the index of the table access with the length of the table. This efficiently prevents the attacker from speculatively bypassing the bounds check.

Swivel does not check the indirect jump targets beyond what Cranelift does. At the language level, Wasm already guarantees that the targets of indirect jumps (i.e., the entries in indirect call tables and switch tables) can only be the tops of functions or switch-case targets. Compiled, these correspond to the start of Swivel linear blocks. Thus, an attacker can only train the BTB with entries that point to linear blocks, which, by construction, are safe to execute in any context.

Protecting returns Wasm's execution stack—in particular, return addresses on the stack—cannot be corrupted during sequential execution. Unfortunately, this does not hold speculatively: An attacker can write to the stack (e.g., with a buffer overflow) and speculatively execute a return instruction which will divert the control flow to their location of choice. Swivel ensures that return addresses on the stack cannot be corrupted as such, even speculatively. We do this using a *separate stack* or *shadow stack* [8], as we detail below.

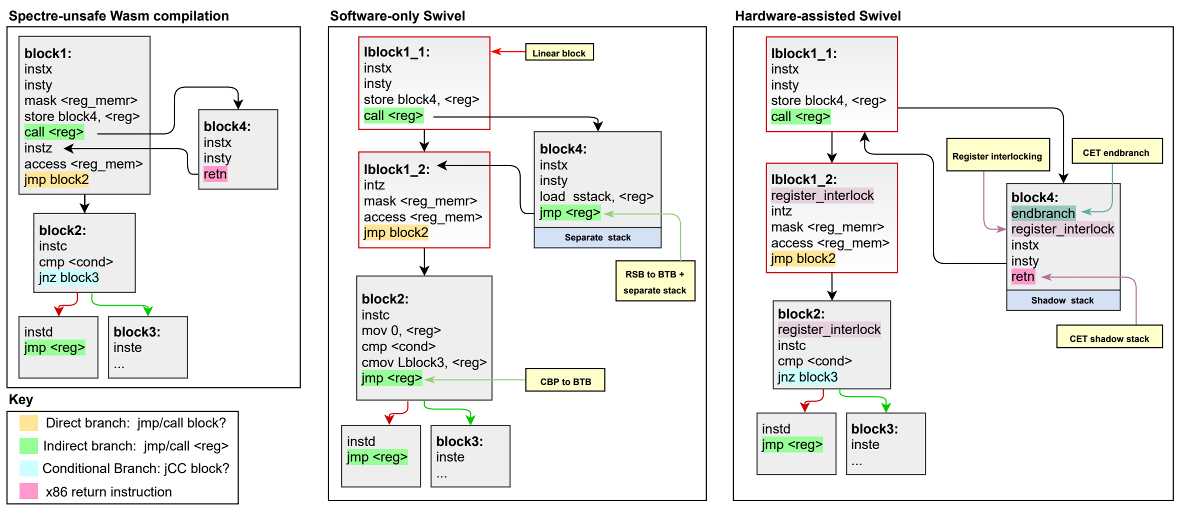


Figure 4: Swivel hardens Wasm against spectre via compiler transformations. In Swivel-SFI, we convert basic blocks to linear blocks. Each linear block (e.g., `lblock1_1` and `lblock1_2`) maintains local security guarantees for speculative execution. Then, we protect the backward edge (`block4`) by replacing the return instructions and using a separate return stack. To eliminate poisoning attacks, in the deterministic version of Swivel-SFI, we further encode conditional branches as indirect jumps. In Swivel-CET, we similarly use linear blocks, but we allow return instruction, and protect returns using the hardware shadow stack. To reduce BTB flushes, we additionally use Intel® CET's `endbranch` to ensure that targets of indirect branches land at the beginning of linear blocks. In the deterministic version, we avoid BTB flushing and instead use register interlocking to prevent leakage on misspeculated paths.

3.2 Swivel-SFI

Swivel-SFI builds on top of linear blocks to address all three classes of attacks.

3.2.1 Addressing sandbox breakout attacks

Compiling Wasm code to linear blocks eliminates most avenues for breaking out of the sandbox. The only way for an attacker to break out of the sandbox is to speculatively jump into the middle of a linear block. We prevent this with:

The separate stack We protect returns by preserving Wasm's safe return stack during compilation. Specifically, we create a separate stack in a fixed memory location, but outside the sandbox stack and heap, to ensure that it cannot be overwritten by sandboxed code. We replace every call instruction with an instruction sequence that stores the address of the subsequent instruction—the return address—to the next entry in this separate stack. Similarly, we replace every return instruction with a sequence that pops the address off the separate stack and jumps to that location. To catch under- and over-flows, we surround the separate stack with guard pages.

BTB flushing The other way an attacker can jump into the middle of a linear block is via a mispredicted BTB entry. Since all indirect jumps inside a sandbox can only point to the tops of linear blocks, any such entries can only be set via a congruent entry outside any sandbox—i.e., an attacker must orchestrate the host runtime into mistraining a particular jump. We prevent such attacks by flushing the BTB on transitions into and out of the sandbox.[2]

[2]In practice, BTB predictions are not absolute (as discussed in our attacker model), instead they are 32-bit offsets relative to the instruction pointer [94]. To ensure that this does not result in predictions at non linear block boundaries, we restrict the sandbox code size to 4GB.

3.2.2 Addressing sandbox and host poisoning attacks

There are two ways for a malicious sandbox to carry out poisoning attacks: By poisoning CBP or BTB entries. Since we already flush the BTB to address sandbox breakout attacks, we trivially prevent all BTB poisoning. Addressing CBP poisoning is less straightforward. We consider two schemes:

Mitigating CBP poisoning To mitigate CBP-based poisoning attacks, we use ASLR to randomize the layout of sandbox code pages. This mitigation is not sound—it is theoretically possible for an attacker to influence a specific conditional branch outside of the sandbox. As we discuss in Section 3.4, this raises the bar to (at least) that of process isolation: The attacker would would have to (1) de-randomize the ASLR of both their own sandbox and the victim's and (2) find useful gadgets, which is itself an open problem (§7).

Eliminating CBP poisoning Clients that are willing to tolerate modest performance overheads (§5) can opt to eliminate poisoning attacks. We eliminate poisoning attacks by removing conditional branches from Wasm sandboxes altogether. Following [54], we do this by using the `cmov` conditional move instruction to encode each conditional branch as an indirect branch with only two targets (Figure 4).

3.3 Swivel-CET

Swivel-SFI avoids using fences to address Spectre attacks, but ultimately bypasses all but the BTB predictors—and even then we flush the BTB on every sandbox transition. Swivel-CET uses Intel® CET [39] and Intel® MPK [39] to restore the use of the CBP and RSB, and avoid BTB flushing.[3]

3.3.1 Addressing sandbox breakout attacks

Like Swivel-SFI, we build on linear blocks to address sandbox breakout attacks (Figure 4). Swivel-CET, however, prevents an attacker from speculatively jumping into the middle of a linear block using:

The shadow stack Swivel-CET uses Intel® CET's *shadow stack* to protect returns. Unlike Swivel-SFI's separate stack, the shadow stack is a hardware-maintained stack, distinct from the ordinary data stack and inaccessible via standard load and store instructions. The shadow stack allows us to use call and return instructions as usual—the CPU uses the shadow stack to check the integrity of return addresses on the program stack during both sequential and speculative execution.

Forward-edge CFI Instead of flushing the BTB, Swivel-CET uses Intel® CET's coarse-grained control flow integrity (CFI) [1] to ensure that sandbox code can only jump to the top of a linear block. We do this by placing an `endbranch` instruction at the beginning of every linear block that is used as an indirect target (e.g., the start of a function that is called indirectly). During speculative execution, if the indirect branch predictor targets an instruction other than an `endbranch` (e.g., inside the host runtime), the CPU stops speculating [39].

Conditional BTB flushing When using ASLR to address sandbox poisoning attacks, we still need to flush the BTB on transitions into each sandbox. Otherwise, one sandbox could potentially jump to a linear block in another sandbox. Our deterministic approach to sandbox poisoning (described below), however, eliminates BTB flushes altogether.

3.3.2 Addressing host poisoning attacks

To prevent host poisoning attacks, Swivel-CET uses Intel® MPK. Intel® MPK exposes new user mode instructions that allow a process to partition its memory into sixteen linear regions and to selectively enable/disable read/write access to any of those regions. Swivel-CET uses only two of these protection domains—one for the host and one shared by all sandboxes—and switches domains during the transitions between host and sandbox. When the host creates a new sandbox, Swivel-CET allocates the heap memory for the new sandbox with the sandbox protection domain, and then relinquishes its own access to that memory so that it is no longer accessible by the host. This prevents host poisoning attacks by ensuring that the host cannot be coerced into leaking secrets from another sandbox. We describe how we safely copy data across the boundary later (§4).

3.3.3 Addressing sandbox poisoning attacks

By poisoning CBP or BTB entries, a malicious sandbox can coerce a victim sandbox into executing a gadget that leaks sensitive data. As with Swivel-SFI, we consider both a probabilistic and deterministic design to addressing these attacks. Since the probabilistic approach is like Swivel-SFI's, we describe only the deterministic design.

Preventing leaks under poisoned execution Swivel-CET does *not* eliminate cross-sandbox CBP or BTB poisoning. Instead, we ensure that a victim sandbox cannot be coerced into leaking data via the cache when executing a mispredicted path. To leak secrets through the cache, the attacker must maneuver the secret data to a gadget that will use it as an offset into a memory region. In Cranelift, any such gadget will use the heap, as stack memory is always accessed at constant offsets from the stack pointer (which itself cannot be directly assigned). We thus need only prevent leaks that are via the Wasm heap—we do this using *register interlocks*.

Register interlocking Our register interlocking technique tracks the control flow of a Wasm program and prevents it from accessing its stack or heap when the speculative path diverges from the sequential path. We first assign each non-trivial linear block a unique 64-bit *block label*. We then calculate the expected block label of every direct or indirect branch and assign this value to a reserved *interlock register* prior to branching. At the beginning of each linear block, we check that the value of the interlock register corresponds to the static block label using `cmov` instructions. If the two do not match, we zero out the heap base register as well as the stack register. Finally, we unmap pages from the address space to ensure that any access from a zero heap or stack base will fault—and thus will not affect cache state.

The register interlock fundamentally introduces a data dependency between memory operations and the resolution of control flow. In doing so, we prevent any memory operations that would result in cache based leaks, but do not prevent all speculative execution. In particular, any arithmetic operations may still be executed speculatively. This is similar to hardware taint tracking [93], but enforced purely through compiler changes.

Finally, Wasm also stores certain data (e.g., globals variables and internal structures) outside the Wasm stack or heap. To secure these memory accesses with the register interlock, we introduce an artificial heap load in the address computation for this data.

3.4 Security and performance trade-offs

Swivel offers two design points for protecting Wasm modules from Spectre attacks: Swivel-SFI and Swivel-CET. For each of these schemes we further consider probabilistic (ASLR) and deterministic techniques. In this section, we discuss the performance and security trade-offs when choosing between these various Swivel schemes.

3.4.1 Probabilistic or deterministic?

Table 1 summarizes Swivel's security guarantees. Swivel's deterministic schemes eliminate Spectre attacks, while the

[3] Appendix A.1 gives a brief introduction to these new hardware features.

Swivel protection and technique	Swivel-SFI		Swivel-CET	
	ASLR	Det	ASLR	Det
Sandbox breakout protections				
- Linear blocks [CBP, BTB, RSB]	✓	✓	✓	✓
- BTB flush in springboard [BTB]	✓	✓		
- Separate control stack [RSB]	✓	✓		
- CET endbranch [BTB]			✓	✓
- CET shadow stack [RSB]			✓	✓
Sandbox poisoning protections				
- BTB flush in springboard [BTB]	✓	✓	✓	
- Code page ASLR [CBP]	✓		✓	
- Direct branches to indirect [CBP]		✓		
- Register interlock [CBP, BTB]				✓
Host poisoning protections				
- Separate control stack [RSB]	✓	✓		
- Code page ASLR [CBP]	✓			
- BTB flush in trampoline [BTB]	✓	✓		
- Direct branches to indirect [CBP]		✓		
- Two domain MPK [CBP]			✓	✓

Table 2: Breakdown of Swivel's individual protection techniques which, when combined, address the thee different class of attacks on Wasm (§2.3). For each technique we also list (in brackets) the underlying predictors.

probabilistic schemes eliminate Spectre attacks that exploit the BTB and RSB, but trade-off security for performance when it comes to the CBP (§5): Our probabilistic schemes only *mitigate* Spectre attacks that exploit the CBP.

To this end, (probabilistic) Swivel hides branch offsets by randomizing code pages. Previously, similar fine-grain approaches to address randomization have been proposed to mitigate attacks based on return-oriented programming [15, 22]. Specifically, when loading a module, Swivel copies the code pages of the Wasm module to random destinations, randomizing all but the four least significant bits (LSBs) to keep 16-byte alignment. This method is more fine-grained than page remapping, which would fail to randomize the lower 12 bits for 4KB instruction pages.

Unfortunately, only a subset of address bits are typically used by hardware predictors. Zhang et. al [94], for example, found that only the 30 LSBs of the instruction address are used as input for BTB predictors. Though a similar study has not been conducted for the CBP, if we pessimistically assume that 30 LSBs are used for prediction then our randomization offers at least 26 bits of entropy. Since the attacker must derandomize both their module and the victim module, this is likely higher in practice.

As we show in Section 5, the ASLR variants of Swivel are faster than the deterministic variants. Using code page ASLR imposes less overhead than the deterministic techniques (summarized in Table 2). This is not surprising: CBP conversion (in Swivel-SFI) and register interlocking (in Swivel-CET) are the largest sources of performance overhead.

For many application domains, this security-performance trade-off is reasonable. Our probabilistic schemes use ASLR only to mitigate sandbox poisoning attacks—and unlike sandbox breakout attacks, these attacks are significantly more challenging for an attacker to carry out: They must conduct an out-of-place attack on a specific target while accounting for the unpredictable mapping of the branch predictor. To our knowledge, such an attack has not been demonstrated, even without the additional challenges of defeating ASLR.

Furthermore, on a FaaS platform, these attacks are even harder to pull off, as the attacker has only a few hundred milliseconds to land an attack on a victim sandbox instance before it finishes—and the next victim instance will have entirely new mappings. Previous work suggests that such an attack is not practical in such a short time window [18].

For other application domains, the overhead of the deterministic Swivel variants may yet be reasonable. As we show in Section 5, the average (geometric) overhead of Swivel-SFI is 47.3% and that of Swivel-CET is 96.3%. Moreover, users can choose to use Swivel-SFI and Swivel-CET according to their trust model—Swivel allows sandboxes of both designs to coexist within a single process.

3.4.2 Software-only or hardware-assisted?

Swivel-SFI and Swivel-CET present two design points that have different trade-offs beyond backwards compatibility. We discuss their trade-offs, focusing on the deterministic variants.

Swivel-SFI eliminates Spectre attacks by allowing speculation only via the BTB predictor and by controlling BTB entries through linear blocks and BTB flushing. Swivel-CET, on the other hand, allows the other predictors. To do this safely though, we use register interlocking to create data dependencies (and thus prevent speculation) on certain operations after branches. Our interlock implementation only guards Wasm memory operations—this means that, unlike Swivel-SFI, Swivel-CET only prevents *cache-based* sandbox poisoning attacks. While non-memory instructions (e.g., arithmetic operations) can still speculatively execute, register interlocks sink performance: Indeed, the overall performance overhead of Swivel-CET is higher than Swivel-SFI (§5).

At the same time, Swivel-CET can be used to handle a more powerful attacker model (than our FaaS model). First, Swivel-CET eliminates poisoning attacks even in the presence of attacker-controlled input. This is a direct corollary of being able to safely execute code with poisoned predictors. Second, Swivel-CET (in the deterministic scheme) is safe in the presence of hyperthreading; our other Swivel schemes assume that hyperthreading is disabled (§3). Swivel-CET allows hyperthreading because it doesn't rely on BTB flushing; it uses register interlocking to eliminate sandbox poisoning attacks. In contrast, our SFI schemes require the BTB to be isolated for the host and each Wasm sandbox—an invariant that may not hold if, for example, hyperthreading interleaves host application and Wasm code on sibling threads.

4 Implementation

We implement Swivel on top of the Lucet Wasm compiler and runtime [30, 60]. In this section, we describe our modifications to Lucet.

We largely implement Swivel-SFI and Swivel-CET as passes in Lucet's code generator Cranelift. For both schemes, we add support for pinned heap registers and add direct `jmp` instructions to create linear block boundaries. We modify Cranelift to harden switch-table and indirect-call accesses: Before loading an entry from either table, we truncate the index to the length of the table (or the next power of two) using a bitwise mask. We also modify Lucet's stack overflow protection: Lucet emits conditional checks to ensure that the stack does not overflow; these checks are rare and we simply use `lfence`s.

We modify the springboard and trampoline transition functions in the Lucet runtime. Specifically, we add a single `lfence` to each transition function since we must disallow speculation from crossing the host-sandbox boundary.

The deterministic defenses for both Swivel-SFI and Swivel-CET—CBP conversion and register interlocks—increase the cost of conditional control flow. To reduce the number of conditional branches, we thus enable explicit loop unrolling flags when compiling the deterministic schemes.[4] This is not necessary for the ASLR-based variants since they do not modify conditional branches. Indeed, the ASLR variants are straightforward modifications to the dynamic library loader used by the Lucet runtime: Since all sandbox code is position independent, we just copy a new sandbox instance's code and data pages to a new randomized location in memory.

We also made changes specific to each Swivel scheme:

Swivel-SFI We augment the Cranelift code generation pass to replace `call` and `return` instructions with the Swivel-SFI separate stack instruction sequences and we mask pointers when they are unspilled from the stack. For the deterministic variant of Swivel-SFI, we also replace conditional branches with indirect jump instructions, as described in Section 3.2.

To protect against sandbox poisoning attacks (§2.3), we flush the BTB during the springboard transition into any sandbox. Since this is a privileged operation, we implement this using a custom Linux kernel module.

Swivel-CET In the Swivel-CET code generation pass, we place `endbranch` instructions at each indirect jump target in Wasm to enable Intel® CET protection. These indirect jump targets include switch table entries and functions which may be called indirectly. We also use this pass to emit the register interlocks for the deterministic variant of Swivel-CET.

We adapt the springboard and trampoline transition functions to ensure that all uses of `jmp`, `call` and `return` conform to the requirements of Intel® CET. We furthermore use these transition functions to switch between the application and sandbox Intel® MPK domains.

Since Intel® MPK blocks the application from accessing sandbox memory, we add primitives that briefly turn off Intel® MPK to copy memory into and out of sandboxes. We implement these primitives using the `rep` instruction prefix instead of branching code, ensuring that the primitives are not vulnerable to Spectre attacks during this window.

Finally, we add Intel® CET support to both the Rust compiler—used to compile Lucet—and to Lucet itself so that the resulting binaries are compatible with the hardware.

[4]For simplicity, we do this in the Clang compiler when compiling applications to Wasm and not in Lucet proper.

5 Evaluation

We evaluate Swivel by asking four questions:

▶ **What is the overhead of Wasm execution? (§5.1)** Swivel's hardening schemes make changes to the code generated by the Lucet Wasm compiler. We examine the performance impact of these changes on Lucet's Sightglass benchmark suite [9] and Wasm-compatible SPEC 2006 [29] benchmarks.

▶ **What is the overhead of transitions? (§5.2)** Swivel modifies the trampolines and springboards used to transition into and out of Wasm sandboxes. The changes vary across our different schemes—from adding `lfence`s, or flushing the BTB during one or both transition directions, to switching Intel® MPK domains. We measure the impact of these changes on transition costs using a microbenchmark.

▶ **What is the end-to-end overhead of Swivel? (§5.3)** We examine the impact of Wasm execution overhead and transition overhead on a webserver that runs Wasm services. We measure the impact of Swivel protections on five different Wasm workloads running on this webserver.

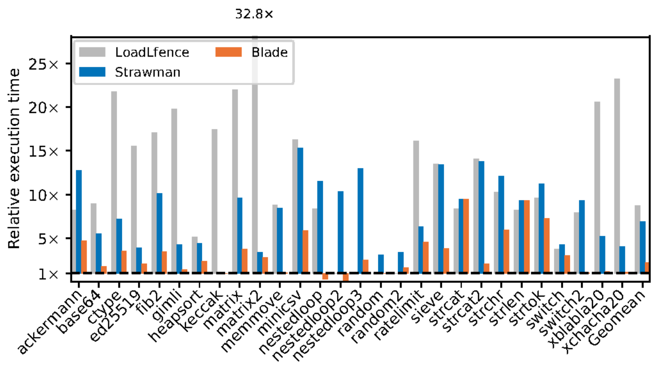
▶ **Does Swivel eliminate Spectre attacks? (§5.4)** We evalute the security of Swivel, i.e., whether Swivel prevents sandbox breakout and poisoning attacks, by implementing several proof-of-concept Spectre attacks.

Machine setup We run our benchmarks on a 4-core, 8-thread Tigerlake CPU software development platform (2.7GHz with a turbo boost of 4.2GHz) supporting the Intel® CET extension. The machine has 16 GB of RAM and runs 64-bit Fedora 32 with the 5.7.0 Linux kernel modified to include Intel® CET support [34]. Our Swivel modifications are applied to Lucet version `0.7.0-dev`, which includes Cranelift version `0.62.0`. We perform benchmarks on standard SPEC CPU 2006, and Sightglass version `0.1.0`. Our webserver macrobenchmark relies on the Rocket webserver version `0.4.4`, and we use `wrk` version `4.1.0` for testing.

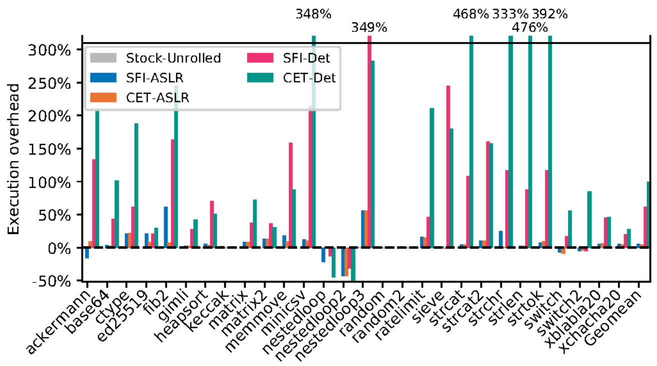
5.1 Wasm execution overhead

We measure the impact of Swivel's Spectre mitigations on Wasm performance in Lucet using two benchmark suites:

▶ The Sightglass benchmark suite [9], used by the Lucet compiler, includes small functions such as cryptographic

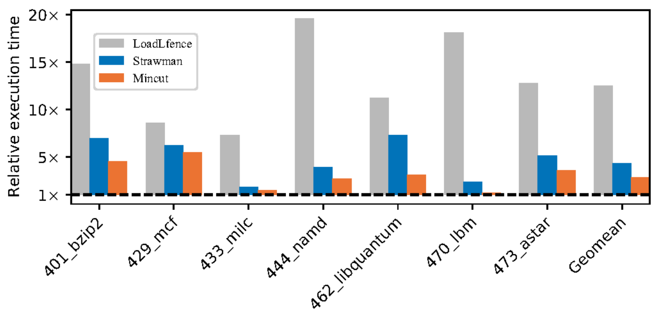


(a) Fence scheme overhead on Sightglass

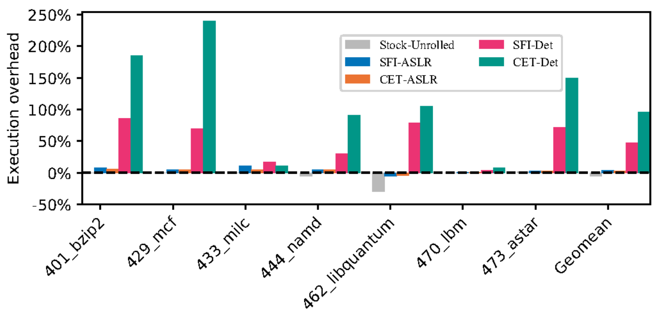


(b) Swivel scheme overhead on Sightglass

Figure 5: Performance overhead of Swivel on the Sightglass benchmarks. (a) On Sightglass, the baseline schemes LoadLfence, Strawman, and Mincut incur geomean overheads of 8.7×, 6.9×, and 2.4× respectively. (b) In contrast, the Swivel schemes perform much better where the ASLR versions of Swivel-SFI and Swivel-CET incur geomean overheads of 5.5% and 4.2% respectively. With deterministic sandbox poisoning mitigations, these overheads are 61.9% and 99.7%.



(a) Fence scheme overhead on SPEC 2006



(b) Swivel scheme overhead on SPEC 2006

Figure 6: Performance overhead of Swivel on SPEC 2006 benchmarks. (a) On SPEC 2006, the baseline schemes LoadLfence, Strawman, and Mincut incur overheads of 7.3×–19.6×, 1.8×–7.3×, and 1.2×–5.4× respectively. (b) In contrast, the Swivel schemes perform much better where the ASLR versions of Swivel-SFI and Swivel-CET incur overheads of at most 10.3% and 6.1% respectively. With deterministic sandbox poisoning mitigations, these overheads are 3.3%–86.1% and 8.0%–240.2% respectively.

primitives (`ed25519`, `xchacha20`), mathematical functions (`ackermann`, `sieve`), and common programming utilities (`heapsort`, `strcat`, `strtok`).

- SPEC CPU 2006 is a popular performance benchmark that includes various integer and floating point workloads from typical programs. We evaluate on only the subset of the benchmarks from SPEC 2006 that are compatible with Wasm and Lucet. This excludes programs written in Fortran, programs that rely on dynamic code rewriting, programs that require more than 4GB of memory which Wasm does not support, and programs that use exceptions, `longjmp`, or multithreading.[5] We note that Swivel does *not* introduce new incompatibilities with SPEC 2006 benchmarks; all of Swivel's schemes are compatible with the same benchmarks as stock Lucet.

Setup We compile both Sightglass and the SPEC 2006 benchmarks with our modified Lucet Wasm compiler and run them with the default settings. Sightglass repeats each test at least 10 times or for a total of 100ms (whichever occurs first) and reports the median runtime. We compare Swivel's performance overhead with respect to the performance of the same benchmarks when using the stock Lucet compiler. For increased measurement consistency on short-running benchmarks, while measuring Sightglass we pin execution to a single core and disable CPU frequency scaling.[6]

Baseline schemes In addition to our comparison against Stock Lucet, we also implement three known Spectre mitigations using `lfence`s and compare against these as a reference point. First, we implement LoadLfence, which places an `lfence` after every load, similar to Microsoft's Visual Studio compiler's "Qspectre-load" mitigation [61]. Next, we implement Strawman, a scheme which restricts code to sequential execution by placing an `lfence` at all control flow targets (at the start of all linear blocks)—this is similar to the Intel compiler's "all-fix-lfence" mitigation [40]. Finally, we implement Mincut, an `lfence` insertion algorithm suggested by Vassena

[5]Some Web-focused Wasm platforms support some of these features. Indeed, previous academic work evaluates these benchmarks on Wasm [42], but non-Web Wasm platforms including Lucet do not support them.

[6]Tests were performed on June 18, 2020; see testing disclaimer A.2.

et al. [86] which uses a min-cut algorithm to minimize the number of required `lfence`s. We further augment Mincut's `lfence` insertion with several of our own optimizations, including (1) only inserting a single `lfence` per linear block; and (3) unrolling loops to minimize branches, as we do for the register interlock scheme and CBP conversions (§3.2.2). Finally, to ensure that unrolling loops does not provide an unfair advantage, we also present results for `Stock-Unrolled`, which is stock Lucet with the same loop unrolling as used in Swivel's schemes.

Results We present the Wasm execution overhead of the various protection options on the Sightglass benchmarks in Figure 5b, and on the SPEC 2006 benchmarks in Figure 6b. The overheads of the ASLR versions of Swivel-SFI and Swivel-CET are small: 5.5% and 4.2% geomean overheads respectively on Sightglass, and at most 10.3% (geomean: 3.4%) and at most 6.1% (geomean: 2.6%) respectively on SPEC. The deterministic versions of Swivel introduce modest overheads: 61.9% and 99.7% on Sightglass, and 3.3%–86.1% (geomean: 47.3%) and 8.0%–240.2% (geomean: 96.3%) on SPEC. All four configurations outperform the baseline schemes by orders of magnitude: Strawman incurs geomean overheads of 6.9× and 4.3× on Sightglass and SPEC respectively, Load-Lfence incurs 8.7× and 12.5× overhead respectively, while Mincut incurs 2.4× and 2.8× respectively.

Breakdown Addressing CBP poisoning (CBP-to-BTB conversion in Swivel-SFI and register interlocks in Swivel-CET) dominates the performance overhead of our deterministic implementations. We confirm this hypothesis with a microbenchmark: We measure the overheads of these techniques individually on stock Lucet (with our loop unrolling flags). We find that the average (geomean) overhead of CBP conversion is 52.9% on Sightglass and 38.5% on SPEC. The corresponding overheads for register interlocks are 93.2% and 53.4%.

Increasing loop unrolling thresholds does not significantly improve the performance of stock Lucet (e.g., the speedup of our loop unrolling on stock Lucet is 0.0% and 5.9% on Sightglass and SPEC, respectively). It does impact the performance of our deterministic Swivel variants though (e.g., we find that it contributes to a 15%-20% speed up). This is not surprising since loop unrolling results in fewer conditional branches (and thus reduces the effect of CBP conversions and register interlocking).

To understand the outliers in Figure 5 and Figure 6, we inspect the source of the benchmarks. Some of the largest overheads in Figure 5 are on Sightglass' string manipulation benchmarks, e.g., `strcat` and `strlen`. These microbenchmarks have lots of *data-dependent loops*—tight loops with data-dependent conditions—that cannot be unrolled at compile-time. Since our register interlocking inserts a data dependence between the pinned heap base register and the loop condition, this prevents the CPU from speculatively executing instructions from subsequent iterations. We believe that similar data-dependent loops are largely the cause for the slowdowns on SPEC benchmarks, including `429.mcf` and `401.bzip2`. Some of the other large overheads in Sightglass (e.g., `fib2` and `nestedloop3`) are largely artifacts of the benchmarking suite: These microbenchmarks test simple constructs like loops—and CBP-to-BTB conversion naturally makes (almost empty) loops slow.

Transition Type	Function Invoke	Callback Invoke
Stock	2.14μs	0.07μs
Swivel-SFI (`lfence` + BTB flush both ways)	4.5μs	1.26μs
Swivel-CET ASLR (`lfence` + BTB flush one way + MPK)	4.08μs	0.79μs
Swivel-CET deterministic (`lfence` + MPK)	2.29μs	0.08μs

Table 3: Time taken for transitions between the application and sandbox—for function calls into the sandbox and callback invocations from the sandbox. Swivel overheads are generally modest, with the deterministic variant of Swivel-CET in particular imposing very low overheads.

5.2 Sandbox transition overhead

We evaluate the overhead of context switching. As described in Section 3, Swivel adds an `lfence` instruction to host-sandbox transitions to mitigate sandbox breakout attacks. In addition to this: Swivel-SFI flushes the BTB during each transition; Swivel-CET, in deterministic mode, switches Intel® MPK domains during each transition; and Swivel-CET, in ASLR mode, flushes the BTB in one direction and switches Intel® MPK domain in each transition.

We measure the time required for the host application to invoke a simple no-op function call in the sandbox, as well as the time required for the sandboxed code to invoke a permitted function in the application (i.e., perform a callback). We compare the time required for Wasm code compiled by stock Lucet with the time required for code compiled with our various protection schemes. We measure the average performance overhead across 1000 such function call invocations.[7] These measurements are presented in Table 3.

First, we briefly note that function calls in stock Lucet take much longer than callbacks. This is because the Lucet runtime has not fully optimized the function call transition, as these are relatively rare compared to callback transitions, which occur during every syscall.

In general, Swivel's overheads are modest, with the deterministic variant of Swivel-CET in particular imposing very low overheads. Flushing the BTB does increase transition costs, but the overall effect of this increase depends on how frequently transitions occur between the application and sandbox. In addition, flushing the BTB affects not only transition performance but also the performance of both the host application and sandboxed code. Fully understanding these overheads requires that we evaluate the overall performance impact on real world applications, which we do next.

[7] Tests were performed on June 18, 2020; see testing disclaimer A.2.

Swivel Protection	Templated HTML				XML to JSON				Change JPEG quality				Check SHA-256			
	ALat	TLat	Tput	Size	ALat	TLat	Tput	Size	ALat	TLat	Tput	Size	ALat	TLat	Tput	Size
Stock (unsafe)	20.8ms	42.1ms	4.81k	3.3MB	186ms	228ms	531	3.2MB	2.23s	2.93s	38.2	2.0MB	424ms	532ms	230	3.6MB
Swivel-SFI ASLR	124ms	137ms	803	3.9MB	213ms	281ms	459	3.8MB	2.31s	2.91s	36.9	2.2MB	449ms	608ms	215	4.2MB
Swivel-SFI Det	34.6ms	80.4ms	2.90k	4.2MB	279ms	322ms	350	4.1MB	3.01s	4.13s	26.4	2.9MB	463ms	575ms	210	4.6MB
Swivel-CET ASLR	111ms	123ms	898	3.4MB	197ms	252ms	498	3.3MB	2.30s	2.88s	37.0	2.0MB	409ms	562ms	234	3.7MB
Swivel-CET Det	28.7ms	66.3ms	3.50k	4.1MB	291ms	328ms	338	4.0MB	2.92s	3.81s	27.5	2.9MB	459ms	570ms	211	4.4MB

Table 4: Average latency (**ALat**), 99% tail latency (**TLat**), average throughput (**Tput**) in requests/second and binary files size (**Size**) for the webserver with different Wasm workloads (1k = 10^3, 1m = 10^6).

Swivel Protection	Image classification			
	ALat	TLat	Tput	Size
Stock (unsafe)	9.67s	13.1s	2.05	34.2MB
Swivel-SFI ASLR	9.78s	13.9s	2.03	34.3MB
Swivel-SFI Det	17.7s	28.3s	1.11	34.7MB
Swivel-CET ASLR	9.82s	12.8s	2.02	34.2MB
Swivel-CET Det	15.7s	24.9s	1.26	34.7MB

Table 5: Average latency (**ALat**), 99% tail latency (**TLat**), average throughput (**Tput**) in requests/second and binary files size (**Size**) for the webserver for a long-running, compute-heavy Wasm workload (1k = 10^3, 1m = 10^6).

5.3 Application overhead

We now evaluate Swivel's end-to-end performance impact on a webserver which uses Wasm to host isolated web services.

Setup For this benchmark, we use the Rocket webserver [69], which can host web services written as Wasm modules. Rocket operates very similarly to webservers used in previous academic papers exploring Wasm modules [28, 77] as well as frameworks used by CDNs such as Fastly. We measure the webserver's performance while hosting five different web services with varying CPU and IO profiles. These services perform the following five tasks respectively: (1) expanding an HTML template; (2) converting XML input to JSON output; (3) re-encoding a JPEG image to change image quality; (4) computing the SHA-256 hash of a given input; and (5) performing image classification using inference on a pretrained neural network. We measure the overall performance of the webserver by tracking the average latency, 99% tail-latency, and throughput for each of the five web services. We also measure the size of the Wasm binaries produced.[8]

Results Tables 4 and 5 show results of the webserver measurements. From the table, we see any of sys's schemes only reduce geomean throughput (across all workloads) between 28.4% and 33.7%. Swivel also modestly increases Wasm binary sizes, particularly with its deterministic schemes, due to additional instructions added for separate stack, CBP-to-BTB, and interlock mechanisms.

For long-running, compute-heavy Wasm workloads such as JPEG re-encoding and image classification, Swivel's performance overhead is dominated by Wasm execution overhead measured in Section 5.1. Thus, on these workloads the ASLR versions of Swivel perform much better than the deterministic versions, as their Wasm execution overhead is lower. On the other hand, for short-running workloads such as templated HTML, we observe that the deterministic schemes outperform the ASLR schemes. This is because Swivel's ASLR implementation must remap and `memcpy` the sandbox code pages during sandbox creation, effectively adding a fixed overhead to each request. For short-running requests, this fixed per-request cost dominates overall overhead. In contrast, Stock Lucet and Swivel's deterministic schemes take advantage of shared code pages in memory to create sandboxes more rapidly, incurring lower overhead on short-running requests.

[8]Tests were performed on June 18, 2020; see testing disclaimer A.2.

5.4 Security evaluation

To evaluate the security of Swivel, we implement several Spectre attacks in Wasm and compile this attack code with both stock Lucet and Swivel. We find that stock Lucet produces code that is vulnerable to Spectre, i.e., our proof of concept attacks (POCs) can be used to carry out both breakout and poisoning attacks, and that Swivel mitigates these attacks.

Attack assumptions Our attacks extend Google's Safeside [24] suite and, like the Safeside POCs, rely on three low-level instructions: The `rdtsc` instruction to measure execution time, the `clflush` instruction to evict a particular cache line, and the `mfence` instruction to wait for pending memory operations to complete. While these instructions are not exposed to Wasm code by default, we expose these instructions to simplify our POCs. Additionally, for cross Wasm module attacks, we manually specify the locations where Wasm modules are loaded to simplify the task of finding partial address collisions in the branch predictor.

Our simplifications are not fundamental and can be removed in an end-to-end attack. Previous work, for example, showed how to construct precise timers [20, 73], and how to control cache contents [87] in environments like JavaScript where these instructions are not directly exposed. The effects of the `mfence` instruction can be achieved by executing `nop` instructions until all memory operations are drained. And, in the style of heap and JIT spraying attacks [78], we can increase the likelihood of partial address collision by deploying hundreds to thousands of modules on the FaaS platform.

POC 1: Sandbox breakout via in-place Spectre-PHT Our first POC adopts the original Spectre-PHT bounds-check bypass attack [49] to Wasm. As mentioned in Section 2.3, in Wasm, indirect function calls are expressed as indices into a function table. Hence, the code emitted for the `call_indirect` instruction performs a bounds check, to ensure that the function index is within the bounds of the table, before performing the lookup and call. By inducing a misprediction on this check, our POC can read beyond the function table boundary and treat the read value as a function pointer. This effectively allows us to jump to any code location and speculatively bypass Wasm's CFI (and thus isolation). We demonstrate this by jumping to a host function that returns bytes of a secret array.

POC 2: Sandbox breakout and poisoning via out-of-place Spectre-BTB Our second POC adopts the out-of-place Spectre-BTB attack of Canella et al. [12] to Wasm. Specifically, we mistrain an indirect jump in a victim or attacker-controlled module by training a congruent indirect jump instruction in another attacker-controlled module. We train the jump to land on a gadget of our choice. To demonstrate the feasibility of a sandbox poisoning attack, we target a double-fetch leak gadget. To demonstrate a sandbox breakout attack, we jump in the middle of a basic block to a memory load, skipping Wasm's heap bounds checks.

POC 3: Poisoning via out-of-place Spectre-RSB Our third POC compiles the Spectre-RSB attack from the Google Safeside project [24] to Wasm. This attack underflows the RSB to redirect speculative control flow. We use this RSB underflow behavior to speculatively "return" to a gadget that leaks module secrets. We run this attack entirely within a single Wasm module. However, on a FaaS platform this attack can be used across modules when the FaaS runtime interleaves the execution of multiple modules, similar to the Safeside cross-process Spectre-RSB attack.

Results We developed our POCs on a Skylake machine (Xeon Platinum 8160) and then tested them on both this machine and the Tiger Lake Intel® CET development platform we used for our performance evaluation. We found that stock Lucet on the Skylake machine was vulnerable to all three POCs while Swivel-SFI, both the ASLR and deterministic versions, were not vulnerable. On the Tiger Lake machine, we found that stock Lucet was vulnerable to POC 3 while Swivel-SFI and Swivel-CET, both the ASLR and deterministic versions, were not. Although the Tiger Lake CPU is documented to be vulnerable to all three Spectre variants [41], we did not successfully reproduce POC 1 and POC 2 on this machine. Getting these attacks to work may require reverse engineering the branch predictors used on these new CPUs. We thus leave the extensions of our POCs to this microarchitecture to future work.

6 Limitations and discussion

In this section, we cover some of the current limitations of Swivel, briefly mention alternate design points, and address the generality of our solutions.

6.1 Limitations of Swivel

We discuss some limitations of Swivel, both in general and for our implementation in particular.

Implementation limitations For this paper, we have simplified some of the implementation details for Swivel-CET to reduce the engineering burden of modifying multiple compiler toolchains and standard libraries while still providing accurate performance evaluations. First, we do not ensure that interlock labels are unique to each linear block, but rather reuse interlock labels; while unique labels are critical for security, previous works have extensively demonstrated the feasibility of assigning unique labels [7]. Our goal was to measure the performance of the instruction sequences for interlock assignment (64-bit conditional moves) and checking (64-bit conditional checks).

Next, when disabling Intel® MPK protections in Swivel-CET (§3.3) in the host calls, we must avoid using indirect branches; while we follow this principle for hostcalls we expose (e.g., when marshaling data for web server requests), we do not modify existing standard library hostcalls. These additional modifications, while straightforward, would require significant engineering effort in modifying the standard library (libc) used by Wasm. Finally, we did not implement the required guard pages in the lower 4GB of memory in our prototype of deterministic Swivel-CET. Prior work [16] has shown how to reserve the bottom 4GB of memory—and that this does not impact performance.

Secretless host Swivel assumes that the host (or runtime) doesn't contain secret information. This assumption is sensible for some applications: in the CDN use case, the CDN part of the process is lightweight and exists only to coordinate with the sandboxes. But not all. As a counter-example, the Firefox web browser currently uses Wasm to sandbox third-party libraries written in C/C++ [21, 65]. We could use Swivel to ensure that Firefox is secure from Spectre attacks conducted by a compromised third-party libraries. To protect secrets in the host (Firefox), we could either place the secrets into a separate Wasm sandbox, or apply one of our proposed CBP protections to the host (e.g., CBP-to-BTB or interlocks).

Hyperthreading The only scheme in Swivel that supports hyperthreading is the deterministic Swivel-CET. Alternately, instead of disabling hyperthreading, Intel suggests relying on single-threaded indirect branch predictors (STIBP) to prevent a co-resident thread from influencing indirect branch predictions [35]. STIBP could allow any Swivel scheme to be used securely with hyperthreading.

6.2 Other leakages and transient attacks

Swivel-CET allows victim code to run with poisoned predictors but prevents exfiltration via the data cache. This, unfortunately, means that attackers may still be able to leak victim data through other microarchitectural channels (e.g., port contention or the instruction cache [49]). Swivel-SFI does not have this limitation; we can borrow techniques from Swivel-SFI to eliminate such leaks (e.g., flushing the BTB).

The CPU's memory subsystem may also introduce other transient execution attacks. Spectre-STL [33] can leak stale data (which may belong to another security domain) before a preceding store could overwrite this data due to speculative dependency checking of the load and the preceding stores. Swivel does not address Spectre-STL. However, Spectre-STL can been mitigated through speculative store bypass disable (SSBD) [37], which imposes a small performance overhead (less than 5% on most benchmarks [53]) and is already enabled by default on most systems.

Meltdown [57] could leak privileged kernel memory from userspace. Variants of the Meltdown attack, e.g., microarchitectural data sampling (MDS), can be used to leak stale data from several microarchitectural elements [11, 38, 63, 72, 84]. Load Value Injection (LVI) exploits the same microarchitectural features as Meltdown to inject data into the microarchitectural state [83]. More recent Intel CPUs (e.g., Tiger Lake) are designed to be resilient against this class of attacks [41], and we believe that these attacks can efficiently be mitigated in hardware. For this reason, recent research into secure speculation and architectural defense against transient execution attacks are mostly focused on the Spectre class of issues [13, 26, 82, 89].

For legacy systems, users should apply the latest microcode and software patches to mitigate Meltdown and variants of MDS [41, 46]. For variants of MDS that abuse hyperthreading on legacy systems, Intel suggests safe scheduling of sibling CPU threads [38]. Since Wasm restricts what instructions are allowed in a Wasm module, this makes some MDS attacks more challenging to execute. For instance, Wasm modules cannot use Intel® TSX transactions or access non-canonical or kernel addresses that are inherent to some of the MDS variants [45]. LVI requires fine-grain control over inducing faults or microcode assists, which is not available at the Wasm level. Some legacy systems may still be vulnerable to LVI; however, the feasibility of LVI attacks outside the Intel SGX environment is an open research question [83].

6.3 Alternate design points for Swivel

We next describe alternate designs for Swivel and discuss the trade-offs of our design choices.

CBP-to-BTB conversion in Swivel-CET Since register interlocking is more expensive than CBP-to-BTB conversion, a reader may wonder whether the deterministic Swivel-CET could more efficiently protect against sandbox poisoning using CBP-to-BTB conversion. Unfortunately, since Swivel-CET does not flush the BTB in both directions, CBP-to-BTB conversion is not sufficient to fully mitigate sandbox poisoning. In addition to CBP-to-BTB conversion, Swivel-CET would also need to use interlocking (without additional performance gain) or flush the BTB both ways. In the latter case, we might as well use Swivel-SFI, as the main advantage of using Intel® CET in Swivel is to avoid flushing the BTB.

Interlocking in Swivel-SFI Likewise, one may wonder about the benefits of using interlock in Swivel-SFI. Unfortunately, for interlock to be useful, it requires hardware support. First, we require Intel® MPK to ensure that a sandbox can't confuse the host into accessing (and then leaking) another sandbox's data. Second, we require the Intel® CET `endbranch` instruction to ensure that the sandbox cannot use BTB entries leftover from the host.

Partitioning shared resources A different approach to addressing Spectre would be to partition differ hardware structures to ensure isolation. For example, for the CBP, one approach would be to exploit the indexing mechanism of branch predictors such that each sandbox uses an isolated portion of the CBP. Unfortunately doing this on existing CPUs is hard: superscalar CPUs use complex predictors with multiple indexing functions, and without knowledge of the underlying microarchitecture, we were unable to experimentally find a way to partition the CBP.

Alternately, we could mitigate host poisoning—or even sandbox poisoning—attacks by partitioning CPU cores, and preventing an attacker from running on the same core as their victim. This approach protects against sandbox poisoning and host poisoning, since branch predictors are per-physical-core. We tried this. Specifically, we implemented this mitigation in the host poisoning context and measured its performance. Unfortunately, requiring a core transition during every springboard and trampoline is detrimental to performance, and this scheme was not competitive with our chosen implementation.

6.4 Generalizing Swivel

Swivel's techniques are not specific to the Lucet compiler or runtime. Our techniques can be applied to other Wasm compilers and runtimes, including the just-in-time Wasm compilers used in the Chrome and Firefox browsers.

Our techniques can also be adopted to other software-based fault isolation (SFI) compilers [80]. Adopting Swivel to the Native Client (NaCl) compiler [74, 92], for instance, only requires only a handful of changes. For example, we wouldn't even need to add linear blocks: NaCl relies on instruction bundles—32-byte aligned blocks of instructions—which are more restrictive than our linear blocks (and satisfy our linear block invariants).

More generally, Swivel can be adopted to other sandboxed languages and runtimes. JavaScript just-in-time compilers are a particularly good fit. Though JavaScript JITs are more complex than Wasm compilers, they share a similar security model (e.g., JavaScript in the browser is untrusted) and, in some cases, even share a common compilation pipeline. For example, Cranelift—the backend used by Lucet and Swivel—was designed to replace Firefox's JavaScript and Wasm backend implementations [23], and thus could transparently benefit from our mitigations. Beyond Cranelift, we think that adopting our linear blocks and code page ASLR is relatively simple (e.g., compared to redesigning the browser to deal with Spectre) and could make JavaScript Spectre attacks significantly more difficult.

6.5 Implementation bugs in Wasm

Lehmann et al. [55] showed that some Wasm compilers and runtimes, like prior SFI toolchains [80], contain implementation bugs.[9] For example, they showed that some Wasm runtimes fail to properly separate the stack and heap. Though they did not identify such bugs in Lucet, these classes of bugs are inevitable—and, while identifying such bugs is important, this class of bugs is orthogonal and well-understood in the SFI literature (and addressed, for example, by VeriWasm [44]). We focus on addressing Spectre attacks, which can fundamentally undermine the guarantees of even bug-free Wasm toolchains.

6.6 Future work

Swivel's schemes can benefit from extensions to compiler toolchains as well as hardware to both simplify its mitigations and improve performance. We briefly discuss some possible extensions and their benefits below.

6.6.1 Compiler toolchain extensions

We describe two performance optimizations for the Swivel-CET deterministic scheme, and a way to improve the security of Swivel's ASLR schemes.

Data dependent loops As discussed in Section 5.1, the Swivel-CET deterministic scheme imposes the greatest overheads in programs with data-dependent loops—e.g., programs that iterate over strings or linked lists (which loop until they find a null element). Swivel effectively serializes iterations of such data-dependent loops. We expect that many other Spectre mitigation (see Section 7), like speculative taint tracking [93], would similarly slow down such programs.

One way to speed up such code is to replace the data-dependent loops with a code sequence that first counts the expected number of iterations (N), executes an `lfence`, and then runs the original loop body for N iterations. This would introduce only a single stall in the loop and eliminate the serialization between loop iterations.

Compiler secret tracking Swivel currently assumes all locations in memory contain potentially secret data. However, several works (e.g., [89]) have proposed tracking secrets in compiler passes. This information can be used to optimize the Swivel-CET deterministic scheme. In particular, any public memory access can be hoisted above the register interlock to allow the memory to be accessed (and "leaked") speculatively.

[9]They also show that C memory safety bugs are still present within the Wasm sandbox—this class of bugs is orthogonal and cannot alone be used to to bypass Wasm's isolation guarantees.

Software diversity Swivel's ASLR variants randomize code pages. We could additionally use software diversity to increase the entropy of our probabilistic schemes [19, 31]. Software diversity techniques (e.g., `nop` insertion) are cheap [32], and since they do not affect the behavior of branches, they can be used to specifically mitigate out-of-place Spectre-BTB and Spectre-PHT attacks.

6.6.2 Hardware extensions

Hardware extensions can make Swivel faster and simpler.

CBP flushing Swivel-SFI schemes rely on ASLR or CBP-to-BTB conversion to protect the CBP. However, hardware support for CBP flushing could significantly speed up Swivel. Alternatively, hardware support for tagging predictor state (e.g., host code and sandbox code) would allow Swivel to isolate the CBP without flushing.

Dedicated interlock instructions The register interlocking used in deterministic Swivel-CET requires several machine instructions in each linear block in order to assign and check labels. Dedicated hardware support for these operations could reduce code bloat.

Explicit BTB prediction range registers The Swivel-CET deterministic scheme allocates unique 64-bit labels to each linear block, which do not overlap across sandbox instances. We could simplify and speed up this scheme with a hardware extension that can be used to limit BTB predictions to a range of addresses. With such an extension, Swivel could set the prediction range during each transition into the sandbox (to the sandbox region) and ensure that the BTB could only predict targets inside the sandbox code pages. This would eliminate out-of-place BTB attacks—and, with linear blocks, it would eliminate breakout attacks in Wasm. Finally, this would reduce code size: it would allow us to to reduce block labels to, for example, 16 bits (since we only need labels to be unique within the sandbox).

7 Related work

We give an overview of related work on mitigating Spectre attacks by discussing microarchitectural proposals, software-based approaches for eliminating Spectre gadgets, and previous approaches based on CFI or Intel® MPK.

Thwarting covert channels Several works [5, 47, 48, 70, 91] propose making microarchitectural changes to block, isolate, or remove the covert channels used to transfer transient secrets to architectural states. For example, *SafeSpec* [47] proposes a speculation-aware memory subsystem which ensures that microarchitectural changes to the cache are not committed until predictions are validated. Similarly, *CleanupSpec* [70] proposes an undo logic for the cache state. Although these approaches remove the attacker's data leakage channel, they do not address the root cause of Spectre vulnerabilities. In contrast, Swivel works with no hardware changes.

Safe speculation Intel has introduced hardware support to mitigate Spectre-BTB across separate address spaces [35, 41]. Specifically, the Indirect Branch Predictor Barrier (IBPB) allows the BTB to be cleared across context switches, while Single Thread Indirect Branch Predictors (STIBP) ensure that one thread's BTB entries will not be affected by the sibling hyperthread. These mitigations can be used by the OS as a coarse-grained mechanism for safe speculation, but only apply to Spectre-BTB and have not been widely adopted due to performance overhead [51].

Other works propose microarchitectural changes to allow the software to control speculation for security-critical operations [81, 90] or certain memory pages [56, 71]. Separately, STT [93] proposes speculative taint tracking within the microarchitecture. However, unlike Swivel, these approaches require significant hardware changes and do not offer a way to safely run code on existing CPUs.

Eliminating Spectre gadgets Another way to mitigate Spectre attacks is by inserting a barrier instruction (e.g., `lfence`), which blocks speculative execution [2, 36, 61]. However, as we evaluated in Section 5.1, insertion of `lfence` has a performance impact on the entire CPU pipeline and undercuts the performance benefit of out-of-order and speculative execution. In contrast, Swivel makes little to no use of `lfence`.

An optimized approach is to replace control flow instructions with alternate code sequences that are safe to execute speculatively. For instance, speculative load hardening (SLH) replaces conditional bounds checks with an arithmetized form to avoid Spectre-PHT [13]. Indeed, Swivel uses SLH to protect the bounds checks for indirect call tables and switch tables (§3.1). Alternatively, Oleksenko et al. [67] propose inserting artificial data dependencies between secret operations and pipeline serialization instructions. Finally, the *retpoline* technique [82] replaces indirect branches with a specific code sequence using the `ret` instruction to avoid Spectre-BTB. To reduce the overhead of such code transformations, researchers have proposed several techniques to automatically locate Spectre gadgets [14, 26, 89] and apply mitigations to risky blocks of code. However, these techniques have to handle potential false positives or negatives; in contrast, Swivel focuses on defending against all possible Spectre attacks from untrusted code by applying compile-time mitigations.

Speculative CFI *SpecCFI* [51] has proposed hardware support for speculative and fine-grained control-flow integrity (CFI), which can be used to protect against attacks on indirect branches. In comparison, Swivel-CET uses Intel® CET, which only supports coarse-grained CFI with speculative guarantees [75]. *Venkman* [76] uses a technique similar to Swivel's linear blocks to ensure that indirect branches always reach a barrier instruction (e.g., `lfence`) by applying alignment to bundles similar to classical software fault isolation [88]. In contrast, Swivel is a fence-free approach that preserves the performance benefits of speculative execution.

Intra-process isolation using Intel® MPK Jenkins et al. [43] propose to provide intra-process Spectre protection using Intel® MPK. They use Intel® MPK to create separate isolation domains and use the relationship between the code and secret data to limit speculative accesses. However, since Intel® MPK only provides 16 domains, relying fully on Intel® MPK to isolate many sandbox instances is infeasible for the CDN Wasm use case we consider.

8 Conclusion

This work proposes a framework, Swivel, which provides strong in-memory isolation for Wasm modules by protecting against Spectre attacks. We describe two Swivel designs: Swivel-SFI, a software-only approach which provides mitigations compatible with existing CPUs, and Swivel-CET, which leverages Intel® CET and Intel® MPK. Our evaluation shows that versions of Swivel using ASLR incur low performance overhead (at most 10.3% on compatible SPEC 2006 benchmarks), demonstrating that Swivel can provide strong security guarantees for Wasm modules while maintaining the performance benefits of in-process sandboxing.

Acknowledgment

We thank Johnnie Birch, Jonathan Foote, Dan Gohman, Pat Hickey, Tyler McMullen, Jan de Mooij, Vedvyas Shanbhogue, Jared Stark, Luke Wagner, and Andy Wortman for insightful discussions. We thank Devdatta Akhawe and the anonymous reviewers for their valuable comments for improving the quality of this paper. We would also like to thank Hongjiu Lu and Yu-cheng Yu for their support on the Intel® CET infrastructure. This work was supported in part by gifts from Cisco, Fastly, Mozilla, and by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA, by the NSF under grant numbers CCF-1918573, CNS-1814406, CAREER CNS-2048262, and by NSF/Intel under grant number CCF-1823444.

References

[1] M. Abadi, M. Budiu, Ú. Erlingsson, and J. Ligatti. Control-flow integrity principles, implementations, and applications. *TISSEC*, 2009.

[2] AMD. Software techniques for managing speculation on AMD processors. http://developer.amd.com/wp-content/resources/Managing-Speculation-on-AMD-Processors.pdf, 2018.

[3] AMD. Speculation behavior in AMD micro-architectures. https://www.amd.com/system/files/documents/security-whitepaper.pdf, 2019.

[4] Apple. About speculative execution vulnerabilities in ARM-based and Intel CPUs. https://support.apple.com/en-us/HT208394, 2018.

[5] K. Barber, A. Bacha, L. Zhou, Y. Zhang, and R. Teodorescu. SpecShield: Shielding speculative data from microarchitectural covert channels. In *PACT*. IEEE, 2019.

[6] J. Bosamiya, B. Lim, and B. Parno. WebAssembly as an intermediate language for provably-safe software sandboxing. PriSC, 2020.

[7] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer. Control-flow Integrity: Precision, Security, and Performance. *CSUR*, 2017.

[8] N. Burow, X. Zhang, and M. Payer. SoK: Shining light on shadow stacks. In *S&P*. IEEE, 2019.

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

[10] Bytecode Alliance. WebAssembly micro runtime. https://github.com/bytecodealliance/wasm-micro-runtime, 2019.

[11] C. Canella, D. Genkin, L. Giner, D. Gruss, M. Lipp, M. Minkin, D. Moghimi, F. Piessens, M. Schwarz, B. Sunar, J. Van Bulck, and Y. Yarom. Fallout: Leaking data on Meltdown-resistant CPUs. In *CCS*. ACM, 2019.

[12] C. Canella, J. Van Bulck, M. Schwarz, M. Lipp, B. von Berg, P. Ortner, F. Piessens, D. Evtyushkin, and D. Gruss. A systematic evaluation of transient execution attacks and defenses. In *SEC*. USENIX, 2019.

[13] C. Carruth. RFC: Speculative load hardening (a Spectre variant #1 mitigation). https://lists.llvm.org/pipermail/llvm-dev/2018-March/122085.html, 2018.

[14] S. Cauligi, C. Disselkoen, K. v. Gleissenthall, D. Tullsen, D. Stefan, T. Rezk, and G. Barthe. Constant-time foundations for the new Spectre era. In *PLDI*. ACM, 2020.

[15] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz. It's a TRaP: Table randomization and protection against function-reuse attacks. In *CCS*. ACM, 2015.

[16] L. Deng, Q. Zeng, and Y. Liu. ISboxing: An instruction substitution based data sandboxing for x86 untrusted libraries. In *IFIP SEC*. Springer, 2015.

[17] Enarx. enarx/enarx Wiki. https://github.com/enarx/enarx/wiki/, 2020.

[18] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev. BranchScope: A new side-channel attack on directional branch predictor. In *ASPLOS*. ACM, 2018.

[19] S. Forrest, A. Somayaji, and D. H. Ackley. Building diverse computer systems. In *Hot Topics in Operating Systems*, 1997.

[20] P. Frigo, C. Giuffrida, H. Bos, and K. Razavi. Grand Pwning Unit: Accelerating microarchitectural attacks with the GPU. In *S&P*. IEEE, 2018.

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

[22] D. Gens, O. Arias, D. Sullivan, C. Liebchen, Y. Jin, and A.-R. Sadeghi. Lazarus: Practical side-channel resilient kernel-space randomization. In *RAID*. Springer, 2017.

[23] D. Gohman. Cranelift in SpiderMonkey. https://github.com/bytecodealliance/wasmtime/blob/main/cranelift/spidermonkey.md, 2018.

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

[25] Google Chrome Team. Site isolation. https://www.chromium.org/Home/chromium-security/site-isolation, 2018.

[26] M. Guarnieri, B. Köpf, J. F. Morales, J. Reineke, and A. Sánchez. SPECTECTOR: Principled detection of speculative information flows. In *S&P*. IEEE, 2020.

[27] 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*. ACM, 2017.

[28] A. Hall and U. Ramachandran. An execution model for serverless functions at the edge. In *IoTDI*. ACM, 2019.

[29] J. L. Henning. SPEC CPU2006 benchmark descriptions. *SIGARCH Computer Architecture News*, 2006.

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

[31] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson. ILR: Where'd my gadgets go? In *S&P*. IEEE, 2012.

[32] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz. Profile-guided automated software diversity. In *CGO*, 2013.

[33] J. Horn. Speculative execution, variant 4: speculative store bypass. https://bugs.chromium.org/p/project-zero/issues/detail?id=1528, 2018.

[34] Intel. CET Linux kernel implementation. https://github.com/hjl-tools/fedora, 2017.

[35] Intel. Intel analysis of speculative execution side channels. https://newsroom.intel.com/wp-content/uploads/sites/11/2018/01/Intel-Analysis-of-Speculative-Execution-Side-Channels.pdf, 2018.

[36] Intel. Speculative execution side channel mitigations. https://software.intel.com/security-software-guidance/api-app/sites/default/files/336996-Speculative-Execution-Side-Channel-Mitigations.pdf, 2018.

[37] Intel. Speculative store bypass / CVE-2018-3639 / INTEL-SA-00115. https://software.intel.com/security-software-guidance/software-guidance/speculative-store-bypass, 2018.

[38] Intel. Deep dive: Intel analysis of microarchitectural data sampling. https://software.intel.com/security-software-guidance/deep-dives/deep-dive-intel-analysis-microarchitectural-data-sampling#SMT-mitigations, 2019.

[39] Intel$^{®}$ 64 and IA-32 architectures software developer's manual, 2020.

[40] Intel® C++ Compiler 19.1 Developer Guide and Reference, 2020.

[41] Intel. Side channel mitigation by product CPU model. https://www.intel.com/content/www/us/en/architecture-and-technology/engineering-new-protections-into-hardware.html, 2020.

[42] A. Jangda, B. Powers, E. D. Berger, and A. Guha. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *ATC*. USENIX, 2019.

[43] I. R. Jenkins, P. Anantharaman, R. Shapiro, J. P. Brady, S. Bratus, and S. W. Smith. Ghostbusting: Mitigating Spectre with intraprocess memory isolation. In *HotSos*, 2020.

[44] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan. Доверяй, но проверяй: SFI safety for native-compiled Wasm. In *NDSS*. Internet Society, 2021.

[45] kernel.org. TAA: TSX asynchronous abort. https://www.kernel.org/doc/html/latest/admin-guide/hw-vuln/tsx_async_abort.html, 2019.

[46] kernel.org. Page Table Isolation (PTI). https://www.kernel.org/doc/html/latest/x86/pti.html, 2020.

[47] K. N. Khasawneh, E. M. Koruyeh, C. Song, D. Evtyushkin, D. Ponomarev, and N. Abu-Ghazaleh. Safespec: Banishing the Spectre of a Meltdown with leakage-free speculation. In *DAC*. IEEE, 2019.

[48] V. Kiriansky, I. Lebedev, S. Amarasinghe, S. Devadas, and J. Emer. DAWG: A defense against cache timing attacks in speculative execution processors. In *MICRO*. IEEE, 2018.

[49] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom. Spectre attacks: Exploiting speculative execution. In *S&P*. IEEE, 2019.

[50] E. M. Koruyeh, K. Khasawneh, C. Song, and N. Abu-Ghazaleh. Spectre returns! Speculation attacks using the return stack buffer. In *WOOT*. USENIX, 2018.

[51] E. M. Koruyeh, S. H. A. Shirazi, K. N. Khasawneh, C. Song, and N. Abu-Ghazaleh. SPECCFI: Mitigating Spectre attacks using CFI informed speculation. In *S&P*. IEEE, 2020.

[52] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song. Code-pointer integrity. In *OSDI*. USENIX, 2014.

[53] M. Larabel. Benchmarking the performance impact of Speculative Store Bypass Disable for Spectre V4 on Intel Core i7. https://www.phoronix.com/scan.php?page=article&item=intel-spectre-ssbd&num=1, 2018.

[54] S. Lee, M.-W. Shih, P. Gera, T. Kim, H. Kim, and M. Peinado. Inferring fine-grained control flow inside SGX enclaves with branch shadowing. In *SEC*. USENIX, 2017.

[55] D. Lehmann, J. Kinder, and M. Pradel. Everything old is new again: Binary security of WebAssembly. In *SEC*. USENIX, 2020.

[56] P. Li, L. Zhao, R. Hou, L. Zhang, and D. Meng. Conditional speculation: An effective approach to safeguard out-of-order execution against Spectre attacks. In *HPCA*. IEEE, 2019.

[57] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, A. Fogh, J. Horn, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg. Meltdown: Reading kernel memory from user space. In *SEC*. USENIX, 2018.

[58] G. Maisuradze and C. Rossow. ret2spec: Speculative execution using return stack buffers. In *CCS*. ACM, 2018.

[59] R. Mcilroy, J. Sevcik, T. Tebbi, B. L. Titzer, and T. Verwaest. Spectre is here to stay: An analysis of side-channels and speculative execution. *arXiv:1902.05178*, 2019.

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

[61] Microsoft. More Spectre mitigations in MSVC. https://devblogs.microsoft.com/cppblog/more-spectre-mitigations-in-msvc/, 2020.

[62] Microsoft Flight Simulator Team. August 20th, 2020 development update. https://www.flightsimulator.com/august-20th-2020-development-update/, 2020.

[63] D. Moghimi, M. Lipp, B. Sunar, and M. Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *SEC*. USENIX, 2020.

[64] Mozilla Wiki. Security/Sandbox. https://wiki.mozilla.org/Security/Sandbox, 2018.

[65] 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 *SEC*. USENIX, 2020.

[66] S. Narayan, T. Garfinkel, S. Lerner, H. Shacham, and D. Stefan. Gobi: WebAssembly as a practical path to library sandboxing. *arXiv:1912.02285*, 2019.

[67] O. Oleksenko, B. Trach, T. Reiher, M. Silberstein, and C. Fetzer. You shall not bypass: Employing data dependencies to prevent bounds check bypass. *arXiv:1805.08506*, 2018.

[68] C. Reis, A. Moshchuk, and N. Oskov. Site isolation: Process separation for web sites within the browser. In *SEC*. USENIX, 2019.

[69] Rocket. https://rocket.rs/, 2020.

[70] G. Saileshwar and M. K. Qureshi. CleanupSpec: An "undo" approach to safe

speculation. In *MICRO*. IEEE, 2019.

[71] M. Schwarz, M. Lipp, C. Canella, R. Schilling, F. Kargl, and D. Gruss. ConTExT: A generic approach for mitigating Spectre. In *NDSS*. Internet Society, 2020.

[72] M. Schwarz, M. Lipp, D. Moghimi, J. Van Bulck, J. Stecklina, T. Prescher, and D. Gruss. ZombieLoad: Cross-privilege-boundary data sampling. In *CCS*. ACM, 2019.

[73] M. Schwarz, C. Maurice, D. Gruss, and S. Mangard. Fantastic timers and where to find them: High-resolution microarchitectural attacks in JavaScript. In *FC*. Springer, 2017.

[74] D. Sehr, R. Muth, C. Biffle, V. Khimenko, E. Pasko, K. Schimpf, B. Yee, and B. Chen. Adapting software fault isolation to contemporary CPU architectures. In *SEC*. USENIX, 2010.

[75] V. Shanbhogue, D. Gupta, and R. Sahita. Security analysis of processor instruction set architecture for enforcing control-flow integrity. In *HASP*, 2019.

[76] Z. Shen, J. Zhou, D. Ojha, and J. Criswell. Restricting control flow during speculative execution with Venkman. *arXiv:1903.10651*, 2019.

[77] S. Shillaker and P. Pietzuch. FAASM: Lightweight isolation for efficient stateful serverless computing. In *ATC*. USENIX, 2020.

[78] A. Sintsov. JIT-spray Attacks & Advanced Shellcode. In *HITBSecConf Amsterdam*, 2010.

[79] L. Sneff. Nebulet. https://github.com/nebulet/nebulet, 2018.

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

[81] M. Taram, A. Venkat, and D. Tullsen. Context-sensitive fencing: Securing speculative execution via microcode customization. In *ASPLOS*. ACM, 2019.

[82] P. Turner. Retpoline: a software construct for preventing branch-target-injection. https://support.google.com/faqs/answer/7625886, 2018.

[83] J. Van Bulck, D. Moghimi, M. Schwarz, M. Lipp, M. Minkin, D. Genkin, Y. Yuval, B. Sunar, D. Gruss, and F. Piessens. LVI: Hijacking transient execution through microarchitectural load value injection. In *S&P*. IEEE, 2020.

[84] S. van Schaik, A. Milburn, S. Österlund, P. Frigo, G. Maisuradze, K. Razavi, H. Bos, and C. Giuffrida. RIDL: Rogue in-flight data load. In *S&P*. IEEE, 2019.

[85] K. Varda. Introducing Cloudflare Workers: Run JavaScript service workers at the edge. https://blog.cloudflare.com/introducing-cloudflare-workers/, 2017.

[86] M. Vassena, C. Disselkoen, K. V. Gleissenthall, S. Cauilgi, R. G. Kici, R. Jhala, D. Tullsen, and D. Stefan. Automatically eliminating speculative leaks with Blade. In *POPL*. ACM, 2021.

[87] P. Vila, B. Köpf, and J. F. Morales. Theory and practice of finding eviction sets. In *S&P*. IEEE, 2019.

[88] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Efficient software-based fault isolation. In *SOSP*. ACM, 1993.

[89] G. Wang, S. Chattopadhyay, I. Gotovchits, T. Mitra, and A. Roychoudhury. oo7: Low-overhead defense against Spectre attacks via binary analysis. *TSE*, 2019.

[90] O. Weisse, I. Neal, K. Loughlin, T. F. Wenisch, and B. Kasikci. NDA: Preventing speculative execution attacks at their source. In *MICRO*. IEEE, 2019.

[91] M. Yan, J. Choi, D. Skarlatos, A. Morrison, C. Fletcher, and J. Torrellas. Invisispec: Making speculative execution invisible in the cache hierarchy. In *MICRO*. IEEE, 2018.

[92] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar. Native Client: A sandbox for portable, untrusted x86 native code. In *S&P*. IEEE, 2009.

[93] J. Yu, M. Yan, A. Khyzha, A. Morrison, J. Torrellas, and C. W. Fletcher. Speculative taint tracking (STT): A comprehensive protection for speculatively accessed data. In *MICRO*. IEEE, 2019.

[94] T. Zhang, K. Koltermann, and D. Evtyushkin. Exploring branch predictors for constructing transient execution trojans. In *ASPLOS*. ACM, 2020.

A Appendix

A.1 Brief introduction to CET and MPK

CET Intel® CET is an instruction set architecture extension that helps prevent Return-Oriented Programming and Call/Jmp-Oriented Programming via use of a *shadow stack*, and *indirect branch tracking* (IBT). The shadow stack is a hardware-maintained stack used exclusively to check the integrity of return addresses on the program stack. To ensure the shadow stack cannot be tampered with, it is inaccessible via standard load and store instructions. The IBT allows the enforcement of coarse-grained control flow integrity (CFI) [1] via a branch termination instruction, `endbranch`. Binaries that wish to use IBT place the `endbranch` at all valid indirect jump targets. If an indirect jump instruction lands on any other instruction, the CPU reports a control-flow protection fault. Additionally, the IBT also supports a *legacy bitmap*, which allows programs to demarcate which code pages have IBT checking enabled.

Importantly, Intel® CET guarantees that any shadow stack mismatches observed during speculative execution of `return` instruction immediately halts further speculative execution. Similarly, any indirect jump during speculative execution from an IBT enabled code page to a page with IBT disabled also halts speculation.

MPK Intel® MPK uses four bits in each page-table entry to assign one of sixteen "keys" to any given memory page, allowing for 16 different memory domains. User mode instructions `wrpkru` and `rdpkru` allow setting read and write permissions for each of these domains on a per-thread basis. Intel® MPK thus allows a process to partition its memory and selectively enable/disable read and write access to any of regions without invoking the kernel functions or switching page tables.

Importantly, `wrpkru` does not execute speculatively - memory accesses affected by the PKRU register will not execute (even speculatively) until all prior executions of `wrpkru` have completed execution and updated the PKRU register and are also resistant to Meltdown style attacks [36].

A.2 Testing Disclaimer

Since we use a software development platform provided by Intel, we include the following disclaimer from Intel:

> Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors. Performance tests, such as SYSmark and MobileMark (in this paper SPEC CPU 2006 and Sightglass), are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks. Performance results are based on testing as of dates shown in configurations and may not reflect all publicly available updates. See backup for configuration details. No product or component can be absolutely secure. Your costs and results may vary. Intel technologies may require enabled hardware, software or service activation.