Isolation without Taxation

Near-Zero-Cost Transitions for WebAssembly and SFI

MATTHEW KOLOSICK, UC San Diego, USA
SHRAVAN NARAYAN, UC San Diego, USA
EVAN JOHNSON, UC San Diego, USA
CONRAD WATT, University of Cambridge, UK
MICHAEL LEMAY, Intel Labs, USA
DEEPAK GARG, Max Planck Institute for Software Systems, Germany
RANJIT JHALA, UC San Diego, USA
DEIAN STEFAN, UC San Diego, USA

Software sandboxing or software-based fault isolation (SFI) is a lightweight approach to building secure systems out of untrusted components. Mozilla, for example, uses SFI to harden the Firefox browser by sandboxing third-party libraries, and companies like Fastly and Cloudflare use SFI to safely co-locate untrusted tenants on their edge clouds. While there have been significant efforts to optimize and verify SFI enforcement, context switching in SFI systems remains largely unexplored: almost all SFI systems use *heavyweight transitions* that are not only error-prone but incur significant performance overhead from saving, clearing, and restoring registers when context switching. We identify a set of *zero-cost conditions* that characterize when sandboxed code has sufficient structured to guarantee security via lightweight *zero-cost* transitions (simple function calls). We modify the Lucet Wasm compiler and its runtime to use zero-cost transitions, eliminating the undue performance tax on systems that rely on Lucet for sandboxing (e.g., we speed up image and font rendering in Firefox by up to 29.7% and 10% respectively). To remove the Lucet compiler and its correct implementation of the Wasm specification from the trusted computing base, we (1) develop a *static binary verifier*, VeriZero, which (in seconds) checks that binaries produced by Lucet satisfy our zero-cost conditions, and (2) prove the soundness of VeriZero by developing a logical relation that captures when a compiled Wasm function is semantically well-behaved with respect to our zero-cost conditions. Finally, we show that our model is useful beyond Wasm by describing a new, purpose-built SFI system, `SegmentZero32`, that uses x86 segmentation and LLVM with mostly off-the-shelf passes to enforce our zero-cost conditions; our prototype performs on-par with the state-of-the-art Native Client SFI system.

CCS Concepts: • **Security and privacy → Formal security models**; **Systems security**.

Additional Key Words and Phrases: software fault isolation, sandboxing, WebAssembly, verification

ACM Reference Format:
Matthew Kolosick, Shravan Narayan, Evan Johnson, Conrad Watt, Michael LeMay, Deepak Garg, Ranjit Jhala, and Deian Stefan. 2022. Isolation without Taxation: Near-Zero-Cost Transitions for WebAssembly and SFI. *Proc. ACM Program. Lang.* 6, POPL, Article 27 (January 2022), 30 pages. https://doi.org/10.1145/3498688

Authors' addresses: Matthew Kolosick, UC San Diego, USA, mkolosick@eng.ucsd.edu; Shravan Narayan, UC San Diego, USA, srn002@eng.ucsd.edu; Evan Johnson, UC San Diego, USA, e5johnso@eng.ucsd.edu; Conrad Watt, University of Cambridge, UK, conrad.watt@cl.cam.ac.uk; Michael LeMay, Intel Labs, USA, michael.lemay@intel.com; Deepak Garg, Max Planck Institute for Software Systems, Germany, dg@mpi-sws.org; Ranjit Jhala, UC San Diego, USA, rjhala@eng.ucsd.edu; Deian Stefan, UC San Diego, USA, deian@cs.ucsd.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2022 Copyright held by the owner/author(s).
2475-1421/2022/1-ART27
https://doi.org/10.1145/3498688

1 INTRODUCTION

Memory safety bugs are the single largest source of critical vulnerabilities in modern software. Recent studies found that roughly 70% of all critical vulnerabilities were caused by memory safety bugs [Chromium Team 2020; Miller 2019] and that malicious attackers are exploiting these bugs before they can be patched [Google Project Zero 2021; Metrick et al. 2020]. Software sandboxing — or software-based fault isolation (SFI) — promises to reduce the impact of such memory safety bugs [Tan et al. 2017; Wahbe et al. 1993]. SFI toolkits like Native Client (NaCl) [Yee et al. 2009] and WebAssembly (Wasm) allow developers to restrict untrusted components to their own *sandboxed* regions of memory thereby isolating the damage that can be caused by bugs in these components. Mozilla, for example, uses Wasm to sandbox third-party C libraries in Firefox [Froyd 2020; Narayan et al. 2020]; SFI allows the browser to use libraries like `libgraphite` (font rendering), `libexpat` (XML parsing), `libsoundtouch` (audio processing), and `hunspell` (spell checking) without risking whole-browser compromise due to library vulnerabilities. Others have used SFI to isolate code in OS kernels [Castro et al. 2009; Erlingsson et al. 2006; Herder et al. 2009; Seltzer et al. 1996], databases [Ford 2005; Ford and Cox 2008; Wahbe et al. 1993], browsers [Haas et al. 2017; Lucco et al. 1995; Yee et al. 2009], language runtimes [Niu and Tan 2014; Siefers et al. 2010], and serverless clouds [Gadepalli et al. 2020; McMullen 2020; Varda 2018].

SFI toolkits enforce memory isolation by placing untrusted code into a sandboxed environment within which every memory access is dynamically checked to be safe. For example, NaCl and Wasm toolkits (e.g., Lucet [Bytecode Alliance 2020a] and WAMR [Bytecode Alliance 2020b]) instrument memory accesses to ensure they are within the sandbox region and add runtime checks to ensure that all control flow is confined to the sandboxed paths with instrumented memory accesses. There is a large body of work that ensures the runtime checks are *fast* on different architectures, e.g., x86 [Ford and Cox 2008; McCamant and Morrisett 2006; Payer and Gross 2011; Yee et al. 2009], x86-64 [Sehr et al. 2010], SPARC® [Adl-Tabatabai et al. 1996], and ARM® [Sehr et al. 2010; Zhao et al. 2011; Zhou et al. 2014], as otherwise they incur unacceptable overheads on the code executing in the sandbox. Similarly, there is a considerable literature that establishes that the checks are *correct* [Besson et al. 2019, 2018; Johnson et al. 2021; Kroll et al. 2014; Morrisett et al. 2012], as even a single missing check can let the attacker escape the sandbox.

However, the security and overhead of software sandboxing also crucially depends on the correctness and cost of context switching — the *trampolines* and *springboards* used to transition into and out of sandboxes. Almost all SFI systems, from [Wahbe et al. 1993]'s original SFI implementation to recent Wasm SFI toolkits [Bytecode Alliance 2020a,b], use *heavyweight transitions* for context switching.[1] These transitions (1) switch protection domains by tying into the underlying memory isolation mechanism (e.g., by setting segment registers [Yee et al. 2009], memory protection keys [Hedayati et al. 2019; Vahldiek-Oberwagner et al. 2019], or sandbox context registers [Bytecode Alliance 2020a,b]), and (2) save, scrub, and restore machine state (e.g. the stack pointer, program counter, and callee-save registers) across the boundary. This code is complicated and hard to get right, as it has to account for the particular quirks of different architectures and operating system platforms [Alder et al. 2020]. Consequently, bugs in transition code have led to vulnerabilities in both NaCl and Wasm — from sandbox breakouts [Nacl Issue 1607 2011; Nacl Issue 1633 2011], to information leaks [Nacl Issue 2919 2012; Nacl Issue 775 2010], and application state corruption [Rydgard 2020]. Furthermore, in applications with high application-sandbox context switching rates, the cost of transitions dominates the overall sandboxing overhead. For example, heavyweight transitions prohibitively slowed down font rendering in Firefox, preventing Mozilla from shipping a sandboxed `libgraphite` [Narayan et al. 2020].

[1] The one exception is WasmBoxC [Zakai 2020], discussed in Section 9.

In this paper, we develop the principles and pragmatics needed to implement SFI systems with near-zero-cost transitions, thereby realizing the three-decade-old vision of reducing the cost of SFI context switches to (almost) that of a function call. We do this via five contributions:

1. Formal model of secure transitions (§3). Simply eliminating heavyweight transitions is unsafe, potentially allowing an attacker to escape the SFI sandbox. To understand this threat, our first contribution is the first formal, declarative, and high-level model that elucidates the role of transitions in making SFI secure. Intuitively, our model shows how secure transitions protect the integrity and confidentiality of machine state across the domain transition by providing *well-bracketed* control flow, i.e., ensuring that returns actually return to their call sites.

2. Zero-cost conditions for isolation (§4). Heavyweight transitions provide security by wrapping cross-domain calls and returns to ensure that sandboxed code cannot, for example, read secret registers or tamper with the stack pointer. While this wrapping is necessary when sandboxing arbitrary code, our insight is these wrappers can be made redundant when the code enjoys additional structure, not dissimilar to the additional structure typically imposed by most SFI systems to ensure memory isolation. For example, NaCl uses *coarse-grained* control-flow integrity (CFI) to restrict the sandbox's control flow to its own code region [Haas et al. 2017; Tan et al. 2017; Yee et al. 2009].

We concretize this insight via our second contribution, a precise definition of *zero-cost conditions* that guarantee that sandboxed code can safely use zero-cost transitions. In particular, we show that transitions can be eliminated when sandboxed code follows a *type-directed* CFI discipline, has well-bracketed control flow, enforces local state (stack and register) encapsulation, and ensures registers and stack slots are initialized before use. Our notion of zero-cost conditions is inspired, in part, by techniques that use type- and memory-safe languages to isolate code via language-level enforcement of well-bracketed control flow and local state encapsulation [Grimmer et al. 2015; Hunt and Larus 2007; Maffeis et al. 2010; Mettler et al. 2010; Miller et al. 2008; Morrisett et al. 1999a]. However, instead of requiring developers to rewrite millions of lines of code in high-level languages [Tan et al. 2017], our zero-cost conditions distill the semantic guarantees provided by high-level languages to allow retrofitting zero-cost transitions in the SFI setting. In other part, our work is inspired by Besson et al. [2018], who define a defensive semantics for SFI that captures a notion of sandboxing via simple function calls with a stack shared between the sandbox and host application. Our work builds on this work by addressing two shortcomings: First, their definition does not account for confidentiality of application data, and implementations based on their system would thus need heavyweight transitions to prevent such attacks. Second, their defensive semantics makes fundamental use of guard zones, which limits the flexibility of the framework. Our definitions of zero-cost transitions have no such limitations and fully realize their goal of defining flexible, secure SFI with zero-cost transitions between application and sandbox.

3. Instantiating the zero-cost model (§5). We demonstrate the retrofitting of zero-cost transitions via our third contribution, an instantiation of our zero-cost model to two SFI systems: Wasm and `SegmentZero32`. Previous work has shown how Wasm can provide SFI by compiling untrusted C/C++ libraries to native code using Wasm as an IR [Bosamiya et al. 2020; Narayan et al. 2020, 2019; Zakai 2020]. We show that Wasm satisfies our zero-cost conditions, and replace the heavyweight transitions used by the industrial Lucet Wasm SFI toolkit with zero-cost transitions. Wasm imposes more structure than required by our zero-cost conditions (and Wasm compilers are still relatively new and slow [Jangda et al. 2019]), so, in order to compare the overhead of our zero-cost model to the still fastest SFI implementation — NaCl [Yee et al. 2009] — we design a new prototype SFI system

(SegmentZero32) that: (1) enforces our zero-cost conditions through LLVM-level transformations, and (2) enforces memory isolation in hardware, using 32-bit x86 segmentation.[2]

4. Verifying security at the binary level (§6). Our fourth contribution is a *static verifier*, VeriZero, that checks whether a potentially malicious binary produced by the Lucet toolkit satisfies our zero-cost conditions. This removes the need to trust the Lucet compiler when, for example, compiling third-party Firefox libraries [Narayan et al. 2020] or untrusted tenant code running on Fastly's serverless cloud [McMullen 2020]. To prove the soundness of VeriZero, we develop a logical relation that captures when a compiled Wasm function is well-behaved with respect to our zero-cost conditions and use it to prove that the checks of VeriZero guarantee that the zero-cost conditions are met. We implement VeriZero by extending VeriWasm [Johnson et al. 2021] and show that in just a few seconds, it can (1) verify sandboxed libraries that ship (or are in the process of being shipped) with Firefox, Wasm-compiled SPEC CPU® 2006 benchmarks, and 100,000 programs randomly generated by Csmith [Yang et al. 2011], and (2) catch previous NaCl and Wasm vulnerabilities (§7.4). VeriZero is being integrated into the Lucet industrial Wasm compiler [Johnson 2021].

5. Implementation and evaluation (§7). Our last contribution is an implementation of our zero-cost sandboxing toolkits, and an evaluation of how they improve the performance of a transition micro-benchmark and two macro-benchmarks — image decoding (libjpeg) and font rendering (libgraphite) in Firefox. First, we demonstrate the potential performance of a purpose-built zero-cost SFI system, by evaluating SegmentZero32 on SPEC CPU® 2006 and our macro-benchmarks. We find that SegmentZero32 imposes at most 25% overhead on SPEC CPU® 2006 (nc), and at most 24% on image decoding and 22.5% on font rendering. These overheads are lower than the state-of-art NaCl SFI system. On the macro-benchmarks, SegmentZero32 even outperforms an idealized SFI system that enforces memory isolation for free but requires heavyweight transitions. Second, we find that zero-cost transitions speed up Wasm-sandboxed image decoding by (up to) 29.7% and font rendering by 10%. The speedup resulting from our zero-cost transitions allowed Mozilla to ship the Wasm-sandboxed libgraphite library in production.

Open source and data. Our code and data will be made available under an open source license.

2 OVERVIEW

In this section we describe the role of transitions in making SFI secure, give an overview of existing heavyweight transitions, and introduce our zero-cost model, which makes it possible for SFI systems to replace heavyweight transitions with simple function calls.

2.1 The Need for Secure Transitions

As an example, consider sandboxing an untrusted font rendering library (e.g., libgraphite) as used in a browser like Firefox:

```
1 void onPageLoad(int* text) {
2   ...
3   int* screen = ...; // stored in r12
4   int* temp_buf = ...;
5   gr_get_pixel_buffer(text, temp_buf);
6   memcpy(screen, temp_buf, 100);
7   ...
8 }
```

[2]While the prevalence of 32-bit x86 systems is declining, it nevertheless still constitutes over 20% of the Firefox web browser's user base (over forty million users) [Mozilla 2021]; SegmentZero32 would allow for high performance library sandboxing on these machines.

This code calls the `libgraphite` `gr_get_pixel_buffer` function to render text into a temporary buffer and then copies the temporary buffer to the variable `screen` to be rendered.

Using SFI to sandbox this library ensures that the browser's memory is isolated from `libgraphite` — memory isolation ensures that `gr_get_pixel_buffer` cannot access the memory of `onPageLoad` or any other parts of the browser stack and heap. Unfortunately, memory isolation alone is not enough: if transitions are simply function calls, attackers can violate the calling convention at the application-library boundary (e.g., the `gr_get_pixel_buffer` call and its return) to break isolation. Below, we describe the different ways a compromised `libgraphite` can do this.

Clobbering Callee-Save Registers. Suppose the `screen` variable in the above `onPageLoad` snippet is compiled down to the register `r12`. In the System V calling convention `r12` is a *callee-saved* register [Lu et al. 2018], so if `gr_get_pixel_buffer` clobbers `r12`, then it is also supposed to restore it to its original value before returning to `onPageLoad`. A compromised `libgraphite` doesn't have to do this; instead, the attacker can poison the register:

```
1  mov r12, ☣
2  ret
```

Since `r12` (`screen`) in our hypothetical example is then used on Line 6 to `memcpy` the `temp_buf` from the sandbox memory, this gives the attacker a write gadget that they can use to hijack Firefox's control flow. To prevent such attacks, we need *callee-save register integrity*, i.e., we must ensure that sandboxed code restores callee-save registers upon returning to the application.

Leaking Scratch Registers. Dually, *scratch registers* can potentially leak sensitive information into the sandbox. Suppose that Firefox keeps a secret (e.g., an encryption key) in a scratch register. Memory isolation alone would not prevent an attacker-controlled `libgraphite` from using uninitialized registers, thereby reading this secret. To prevent such leaks, we need *scratch register confidentiality*.

Reading and corrupting stack frames. Finally, if the application and sandboxed library share a stack, the attacker could potentially read and corrupt data (and pointers) stored on the stack. To prevent such attacks, we need *stack frame encapsulation*, i.e., we need to ensure that sandboxed code cannot access application stack frames.

2.2 Heavyweight Transitions

SFI toolchains — from NaCl [Yee et al. 2009] to Wasm native compilers like Lucet [Bytecode Alliance 2020a] and WAMR [Bytecode Alliance 2020b] — use *heavyweight transitions* to wrap calls and returns and address the aforementioned attacks. These heavyweight transitions are secure transitions. They provide:

1. Callee-save register integrity. The *springboard* — the transition code which wraps calls — saves callee-save registers to a separate stack stored in protected application memory. When returning from the library to the application, the *trampoline* — the code which wraps returns — restores the registers.

2. Scratch register confidentiality. Since any scratch register may contain secrets, the springboard clears *all* scratch registers before transitioning into the sandbox.

3. Stack frame encapsulation. Most (but not all) SFI systems provision separate stacks for trusted and sandboxed code and ensure that the trusted stack is not accessible from the sandbox. The springboard and trampoline account for this in three ways. First, they track the separate stack pointers at each transition in order to switch stacks. Second, the springboard copies arguments passed on the stack to the sandbox stack, since sandboxed code cannot access arguments stored

on the application stack. Finally, the trampoline tracks the actual return address on transition by keeping it in the protected memory, so that the sandboxed library cannot tamper with it.

The Cost of Wrappers. Heavyweight springboards and trampolines guarantee secure transitions but have two significant drawbacks. First, they impose an overhead on SFI — calls into the sandboxed library become significantly more expensive than simple application function calls (§7). Heavyweight transitions conservatively save and clear more state than might be necessary, essentially reimplementing aspects of an OS process switch and duplicating work done by well-behaved libraries. Second, springboards and trampolines must be customized to different platforms, i.e., different processors and calling conventions, and, in extreme cases such as in Vahldiek-Oberwagner et al. [2019], even different applications. Implementation mistakes can — and have [Bartel and Doe 2018; Nacl Issue 1607 2011; Nacl Issue 1633 2011; Nacl Issue 2919 2012; Nacl Issue 775 2010; Native Client team 2009] — resulted in sandbox escape attacks.

2.3 Zero-Cost Transitions

Heavyweight transitions are conservative because they make few assumptions about the structure (or possible behavior) of the code running in the sandbox. SFI systems like NaCl and Wasm *do*, however, impose structure on sandboxed code to enforce memory isolation. In this section we show that by imposing structure on sandboxed code we can make transitions less conservative. Specifically, we describe a set of *zero-cost conditions* that impose *just enough* internal structure on sandboxed code to ensure that it will behave like a high-level, compositional language while maintaining SFI's high performance. SFI systems that meet these conditions can safely elide almost all the extra work done by heavyweight springboards and trampolines, thus moving toward the ideal of SFI transitions as simple, fast, and portable function calls.

Zero-Cost Conditions. We assume that the sandboxed library code is split into functions and that each function has an expected number of arguments. We *formalize* the internal structure required of library code via a *safety monitor* that checks the zero-cost conditions, i.e., the local requirements necessary to ensure that calls-into and returns-from the untrusted library functions are "well-behaved" and, hence, that they satisfy the secure transition requirements.

1. Callee-save register restoration. First, our monitor enforces function-call level adherence to callee-save register conventions: our monitor tracks callee-save state and checks that it has been correctly restored upon returning. Importantly, satisfying the monitor means that application calls to a well-behaved library function do not require a transition which separately saves and restores callee-save registers, since the function is known to obey the standard calling convention.

2. Well-bracketed control-flow. Second, our monitor requires that the library code adheres to well-bracketed return edges. Abstractly, calls and returns should be well-bracketed: when `f` calls `g` and then `g` calls `h`, `h` ought to return to `g` and then `g` ought to return to `f`. However, untrusted functions may subvert the control stack to implement arbitrary control flow between functions. This unrestricted control flow is at odds with compositional reasoning, preventing *local* verification of functions. Further, subverting well-bracketing could enable an attacker to cause `h` to return directly to `f`. Then, even if `h` and `f` both restore their callee-save registers, those of `g` would be left unrestored. Accordingly, we require two properties of the library to ensure that calls and returns are well-bracketed. First, each jump must stay within the same function. This limits inter-function control flow to function calls and returns. Second, the (specification) monitor maintains a "logical" call stack, which is used to ensure that returns go only to the preceding caller.

3. Type-directed forward-edge CFI. Our monitor also requires that library code obeys type-directed forward-edge CFI. That is, for every call instruction encountered during execution, the jump target address is the start of a library function and the arguments passed match those expected

by the called function. This ensures that each function starts from a (statically) known stack shape, preventing a class of attacks where a benign function can be tricked into overwriting other stack frames or hijacking control flow because it is passed too few (or too many) arguments. If this were not the case, a locally well-behaved function that was passed too few arguments could write to a saved register or the saved return address, expecting that stack slot to be the location of an argument.

4. Local state encapsulation. Our monitor establishes *local state encapsulation* by checking that all stack reads and writes are within the current stack frame. This check allows us to *locally*, i.e., by checking each function in isolation, ensure that a library function correctly saves and restores callee-save registers upon entry and exit. To see why local state encapsulation is needed, consider the following idealized assembly function `library_func`:

```
1  library_func:          library_helper:
2    push r12                store sp - 1 := ☣
3    mov r12 ← 1             ret
4    load r1 ← sp - 1
5    add r1 ← r12
6    call library_helper
7    pop r12
8    ret
```

If `library_helper` is called it will overwrite the stack slot where `library_func` saved `r12`, and `library_func` will then "restore" `r12` to the attacker's desired value. Our monitor prohibits such cross-function tampering, thus ensuring that all subsequent reasoning about callee-save integrity can be carried out locally in each function.

5. Confidentiality. Finally, our monitor uses dynamic information flow control (IFC) tracking to define the confidentiality of scratch registers. The monitor tracks how (secret application) values stored in scratch registers flow through the sandboxed code, and checks that the library code does not leak this information. Concretely, our implementations enforce this by ensuring that, within each function's localized control flow, all register and local stack variables are initialized before use.

The individual properties making up our zero-cost conditions are well-known to be beneficial to software security, and their enforcement in low-level code has been extensively studied (§9): our insight — made manifest in the monitor soundness proofs of Section 4.2 — is that in conjunction these conditions are *sufficient* to eliminate heavyweight transitions in SFI systems, which can currently be a source of significant overhead when sandboxing arbitrary code. Indeed, in Section 5.1 we show that the Wasm type system is strict enough to ensure that a Wasm compiler generates native code that already meets these conditions. To increase the trustworthiness of this zero-cost compatible Wasm, in Section 6 we describe a verifier that statically checks that compiled Wasm code meets the zero-cost conditions. In Section 6.4 we describe our proof of soundness for the verifier, proving that the verifier's checks ensure monitor safety and therefore zero-cost security. Further, in Section 5.2 we demonstrate how the zero-cost conditions can be used to design a new SFI scheme by combining hardware-backed memory isolation with existing LLVM compiler passes.

3 A GATED ASSEMBLY LANGUAGE

We formalize zero-cost transitions via an assembly language, SFIasm, that captures key notions of an application interacting with a sandboxed library, focusing on capturing properties of the transitions between the application and sandboxed library.

Code. Figure 1 summarizes the syntax of SFIasm: a Risc-style language with natural numbers ($\mathbb{N}$) as the sole data type. Code (C) and data (M) memory are separated, and, to capture the separation

$$
\begin{array}{rclcl}
 & & n & \in & \mathbb{N} \\
\mathit{Priv} & \ni & p & ::= & \texttt{app} \mid \texttt{lib} \\
\mathit{Val} & \ni & v & ::= & n \\
\mathit{Reg} & \ni & r & ::= & \texttt{r}_n \mid sp \mid pc \\
\mathit{Region} & \ni & k & \in & \mathbb{N} \rightharpoonup \mathbb{N} \\
\mathit{Expr} & \ni & e & ::= & r \mid v \mid e \oplus e \\
\mathit{Command} & \ni & c & ::= & r \leftarrow \texttt{pop}_p \mid \texttt{push}_p\ e \mid \texttt{jmp}_k\ e \mid r \leftarrow \texttt{load}_k\ e \mid \texttt{store}_k\ e := e \mid \\
 & & & & \texttt{gatecall}_n\ e \mid \texttt{gateret} \mid r \leftarrow \texttt{mov}\ e \mid \texttt{call}_k\ e \mid \texttt{ret}_k \\
\mathit{Code} & \ni & C & ::= & \mathbb{N} \rightharpoonup \mathit{Priv} \times \mathit{Command} \\
\mathit{RegVals} & \ni & R & ::= & \mathit{Reg} \to \mathit{Val} \\
\mathit{Memory} & \ni & M & ::= & \mathbb{N} \to \mathit{Val} \\
\mathit{State} & \ni & \Psi & ::= & \texttt{error} \mid \{pc : \mathbb{N}, sp : \mathbb{N}, R : \mathit{RegVals}, M : \mathit{Memory}, C : \mathit{Code}\}
\end{array}
$$

Fig. 1. Syntax

of application code from sandboxed library code, C is an (immutable) partial map from $\mathbb{N}$ to pairs of a privilege (p) (`app` or `lib`) and a command (c), where `app` and `lib` are our *security domains*.

States. Memory is a (total) map from $\mathbb{N}$ to values (v). We assume that the memory is subdivided into disjoint regions (M_p) so that the application and library have separate memory. Each of these regions is further divided into a disjoint heap H_p and stack S_p. We write Ψ to denote the states or machine configurations, which comprise code, memory, and a fixed, finite set of registers mapping register names (r_n) to values, with a distinguished stack pointer (sp) and program counter (pc) register. We write $\Psi(\!|c|\!)_p$ for $\Psi.C(\Psi.pc) = (p, c)$, that is that the current instruction is c in security domain p. We write $\Psi_0 \in \mathit{Program}$ to mean that Ψ_0 is a valid initial program state. The definition of validity varies between different SFI techniques (e.g., heavyweight transitions make assumptions about the initial state of the separate stack).

Gated Calls and Returns. We capture the transitions between the application and the library by defining a pair of instructions $\texttt{gatecall}_n\ e$ and `gateret`, that serve as the *only* way to switch between the two security domains (that is, `call` and `ret` check that the target is in the same security domain). The first, $\texttt{gatecall}_n\ e$, represents a call from the application into the sandbox or a callback from the sandbox to the application with the n annotation representing the number of arguments to be passed. The second, `gateret`, represents the corresponding return from sandbox to application or vice-versa. We leave the reduction rule for both *implementation specific* in order to capture the details of a given SFI system's trampolines and springboards.

Memory Isolation. SFIasm provides abstract mechanisms for enforcing SFI memory isolation by equipping the standard `load`, `store`, `push`, and `pop` with (optional) statically annotated checks. To capture different styles of enforcement we model these checks as partial functions that map a pointer to its new value or are undefined when a particular address is invalid. This lets us, for instance, capture NaCl's coarse grained, dynamically enforced isolation (sandboxed code may read and write anywhere in the sandbox memory) by requiring that all loads and stores are annotated with the check $k(n)|_{n \in M_{\texttt{lib}}} = n$. This captures that NaCl's memory isolation does not remap addresses but traps when an address is outside the sandbox memory region ($M_{\texttt{lib}}$).[3] The rule for `load` below demonstrates the use of these region annotations in the semantics.

Control-Flow Integrity. SFIasm also provides abstract control-flow integrity enforcement via annotations on `jmp`, `call`, and `ret`. These are also enforced dynamically. However, we require that

[3]NaCl implements memory protection differently on different platforms. The 32-bit implementation traps whereas the 64-bit implementation masks addresses. We focus on the former.

$$\frac{\Psi_1 \to \Psi_2 \qquad \Psi_1(\!|c_1|\!)_{p_1} \qquad \Psi_2(\!|c_2|\!)_{p_2} \qquad p_1 = p_2 = p}{\Psi_1 \xrightarrow{p} \Psi_2} \qquad \frac{\Psi \xrightarrow{p} \Psi'}{\Psi \xrightarrow{\square} \Psi'} \qquad \frac{\Psi \xrightarrow{\text{wb}} \Psi'}{\Psi \xrightarrow{\square} \Psi'} \qquad \frac{\Psi \to \Psi_1 \xrightarrow{\square}{}^{*} \Psi_2 \to \Psi' \qquad \Psi(\!|\texttt{gatecall}_n\ i|\!) \qquad \Psi_2(\!|\texttt{gateret}|\!)}{\Psi \xrightarrow{\text{wb}} \Psi'}$$

Fig. 2. Well-Bracketed Transitions

the standard control flow operations remain within their own security domain so that `gatecall` and `gateret` remain the only way to switch security domains.

Operational Semantics. We capture the dynamic behavior via a deterministic small-step operational semantics ($\Psi \longrightarrow \Psi'$). The rules are standard; we show the rule for `load` here:

$$\frac{addr = \mathcal{V}_\Psi(e) \qquad addr' = k(addr) \qquad v = \Psi.M(addr') \qquad R' = \Psi.R[r \mapsto v]}{\Psi(\!|r \leftarrow \texttt{load}_k\ e|\!) \to \Psi^{++}[R := R']}$$

$\mathcal{V}_\Psi(e)$ evaluates the expression based on the register file and Ψ^{++} increments pc, checking that it remains within the same security domain and returning an error otherwise. If the function $k(addr)$ is undefined ($addr$ is not within bounds), the program will step to a distinguished, terminal state `error`. $\Psi(\!|c|\!)$ is simply shorthand for $\Psi(\!|c|\!)_p$ when we do not care about the security domain. Lastly, we do not include a specific halt command, instead halting when pc is not in the domain of C.

3.1 Secure transitions

Next, we use SFIasm to *declaratively* specify high-level properties that capture the intended security goals of transition systems. This lets us use SFIasm both as a setting to study zero-cost transitions and to explore the correctness of implementations of springboards and trampolines. As a demonstrative example we prove that NaCl-style heavyweight transitions satisfy the high-level properties (see the technical appendix [Kolosick et al. 2021]).

Well-Bracketed Gated Calls. SFI systems may allow arbitrary *nesting* of calls into and callbacks out of the sandbox. Thus, it is insufficient to define that callee-save registers have been properly restored by simply equating register state upon entry to the sandbox and the following exit. Instead we make the notion of an entry and its *corresponding* exit precise, by using SFIasm's `gatecall` and `gateret` to define a notion of *well-bracketed gated calls* that serve as the backbone of transition integrity properties. A well-bracketed gated call, which we write $\Psi \xrightarrow{\text{wb}} \Psi'$ (Figure 2), captures the idea that Ψ is a gated call from one security domain to another, followed by running in the new security domain, and then Ψ' is the result of a gated return that balances the gated call from Ψ. This can include potentially recursive but properly bracketed gated calls. Well-bracketed gated calls let us relate the state before a gated call with the state after the *corresponding* gated return, capturing when the library has fully returned to the application.

Integrity. Relations between the states before calling into the sandbox and then after the corresponding return capture SFI transition system *integrity* properties. We identify two key integrity properties that SFI transitions must maintain:

1. Callee-save register integrity requires that callee-save registers are restored after returning from a gated call into the library. This ensures that an attacker cannot unexpectedly modify the private state of an application function.

2. Return address integrity requires that the sandbox (1) returns to the instruction after the `gatecall`, (2) does not tamper with the stack pointer, and (3) does not modify the call stack itself. Together these ensure that an attacker cannot tamper with the application control flow.

These integrity properties are crucial to ensure that the sandboxed library cannot break application invariants. To capture them formally, we first define an abstract notion of integrity across a well-bracketed gated call. This not only allows us to cleanly define the above properties, but also provides a general framework that can capture integrity properties for different architectures.

Specifically, we define an integrity property by a predicate $\mathcal{I} : \mathit{Trace} \times \mathit{State} \times \mathit{State} \to \mathbb{P}$ that captures when integrity is preserved across a call ($\mathbb{P}$ is the type of propositions). The first argument is a trace, a sequence of steps that our program has taken before making the gated call. The next two arguments are the states before and after the well-bracketed gated call. $\mathcal{I}$ defines when these two states are properly related. This leads to the following definition of $\mathcal{I}$-Integrity:

Definition 1 ($\mathcal{I}$-Integrity). *Let $\mathcal{I} : \mathit{Trace} \times \mathit{State} \times \mathit{State} \to \mathbb{P}$. We say that an SFI transition system has $\mathcal{I}$-integrity if $\Psi_0 \in \mathit{Program}$, $\pi = \Psi_0 \to^* \Psi_1$, $\Psi_1 (\!|_|\!)_{\mathsf{app}}$, and $\Psi_1 \xrightarrow{wb} \Psi_2$ imply that $\mathcal{I}(\pi, \Psi_1, \Psi_2)$.*

We instantiate this to define our two integrity properties:

Callee-Save Register Integrity. We define callee-save register integrity as an $\mathcal{I}$-integrity property that requires the callee-save registers' values to be equal in both states:

Definition 2 (Callee-Save Register Integrity). *Let $\mathbb{CSR}$ be the callee-save registers and define $\mathcal{CSR}(_, \Psi_1, \Psi_2) \triangleq \Psi_2.R(\mathbb{CSR}) = \Psi_1.R(\mathbb{CSR})$. If an SFI transition system has $\mathcal{CSR}$-integrity then we say it has callee-save register integrity.*

Return Address Integrity. We specify that the library returns to the expected instruction as a relation between Ψ_1 and Ψ_2, namely that $\Psi_2.pc = \Psi_1.pc + 1$. Restoration of the stack pointer is similarly specified as $\Psi_2.sp = \Psi_1.sp$. Specifying call stack integrity is more involved as Ψ_1 lacks information on where return addresses are saved: they look like any other stack data. Instead, return addresses are defined by the history of calls and returns leading up to Ψ_1, which we capture with the trace argument π. We thus define a function return-address(π) (see the technical appendix [Kolosick et al. 2021]) that computes the locations of return addresses from a trace. The third clause of return address integrity is then that these locations' values are preserved from Ψ_1 to Ψ_2, yielding:

Definition 3 (Return Address Integrity).

$$\begin{aligned}\mathcal{RA}(\pi, \Psi_1, \Psi_2) \triangleq\ & \Psi_2.pc = \Psi_1.pc + 1 \wedge \Psi_2.sp = \Psi_1.sp \\ & \wedge \Psi_2.M(\text{return-address}(\pi)) = \Psi_1.M(\text{return-address}(\pi))\end{aligned}$$

If an SFI transition system has $\mathcal{RA}$-integrity then we say the system has return address integrity.

Confidentiality. SFI systems must ensure that secrets cannot be leaked to the untrusted library, i.e., they must provide *confidentiality*. We specify confidentiality as noninterference, which informally states that "changing secret inputs should not affect public outputs." In the context of library sandboxing, application data is secret whereas library data is non-secret (public).[4] To capture this formally, we pair programs with a confidentiality policy, $\mathbb{C} \in \mathit{State} \rightharpoonup (\mathbb{N} + \mathit{Reg} \to \mathit{Priv})$, that labels all memory and registers as app or `lib` at each gated call into the library. These labels form a lattice: `lib` $\sqsubseteq$ app (non-secret *can* "flow to" secret) and app $\not\sqsubseteq$ `lib` (secret *cannot* "flow to" non-secret).[5]

To prove noninterference, that changing secret data does not affect public (or non-secret) outputs, we need to define public outputs. We over-approximate public outputs as the set of values *exposed* to the application. This includes all arguments to a `gatecall` callback, the return value when

[4] This could also be extended to a setting with mutually distrusting components.

[5] Details can be found in the technical appendix [Kolosick et al. 2021].

$$\begin{array}{rcrcl} \mathit{Val} & \ni & v & ::= & \langle n, p \rangle \\ \mathit{Frame} & \ni & \mathit{SF} & ::= & \{\mathit{base} : \mathbb{N}; \mathit{ret\text{-}addr\text{-}loc} : \mathbb{N}; \mathit{csr\text{-}vals} : \wp(\mathit{Reg} \times \mathbb{N})\} \\ \mathit{Function} & \ni & F & ::= & \{\mathit{instrs} : \mathbb{N} \rightharpoonup \mathit{Command}; \mathit{entry} : \mathbb{N}; \mathit{type} : \mathbb{N}\} \\ \mathit{oState} & \ni & \Phi & ::= & \texttt{oerror} \mid \{\Psi : \mathit{State}; \mathit{funcs} : \mathbb{N} \rightharpoonup \mathit{Function}; \mathit{stack} : [\mathit{Frame}]\} \end{array}$$

Fig. 3. oSFIasm Extended Syntax

returning to the application via `gateret`, and all values stored in the sandboxed library's heap ($H_{\texttt{lib}}$) (which may be referenced by other returned values).

Alas, this is not enough: in a callback, the application may choose to declassify secret data. For instance, a sandboxed image decoding library might, after parsing the file header, make a callback requesting the data to decode the rest of the image. This application callback will then transfer that data (which was previously confidential) to the sandbox, declassifying it in the transfer.

To account for such intentional declassifications, we follow Matos and Boudol [2005] and define confidentiality as *disjoint noninterference*. We use $\Psi =_{\mathbb{C}} \Psi'$ to mean that Ψ and Ψ' agree on all values labeled `lib` by the confidentiality policy, capturing varying secret inputs. We further write $\Psi =_{\texttt{call}\ m} \Psi'$ when Ψ and Ψ' agree on all sandboxed heap values, the program counter, and the m arguments passed to a callback and $\Psi =_{\texttt{ret}} \Psi'$ when Ψ and Ψ' agree on all sandboxed heap values, the program counter, and the value in the return register (written r_{ret}).[6] This lets us formally define noninterference as follows:

Definition 4 (Disjoint Noninterference).
We say that an SFI transition system has the disjoint noninterference property if, for all initial configurations and their confidentiality properties $(\Psi_0, \mathbb{C}) \in \mathit{Program}$, *traces* $\Psi_0 \to^* \Psi_1 \to \Psi_2 \xrightarrow{\texttt{lib}}{}^* \Psi_3 \to \Psi_4$, *where* Ψ_1 *is a gated call into the library* ($\Psi_1 (\!| \texttt{gatecall}_n\ e |\!)_{\texttt{app}}$), *and* $\Psi_3 \to \Psi_4$ *leaves the library and reenters the application* ($\Psi_4 (\!| _ |\!)_{\texttt{app}}$), *and, for all* Ψ_1' *such that* $\Psi_1 =_{\mathbb{C}} \Psi_1'$, *we have that* $\Psi_1' \to \Psi_2' \xrightarrow{\texttt{lib}}{}^* \Psi_3' \to \Psi_4'$, $\Psi_4' (\!| _ |\!)_{\texttt{app}}$, $\Psi_4.pc = \Psi_4'.pc$, *and either (1)* Ψ_3 *is a gated call to the application* ($\Psi_3 (\!| \texttt{gatecall}_m\ e |\!)$ *and* $\Psi_3' (\!| \texttt{gatecall}_m\ e |\!)$) *and* $\Psi_4 =_{\texttt{call}\ m} \Psi_4'$ *or (2)* Ψ_3 *is a gated return to the application* ($\Psi_3 (\!| \texttt{gateret} |\!)$ *and* $\Psi_3' (\!| \texttt{gateret} |\!)$) *and* $\Psi_4 =_{\texttt{ret}} \Psi_4'$.

This definition captures that, for any sequence of executing within the library then returning control to the application, varying confidential inputs does not influence the public outputs and the library returns control to the application in the same number of steps. Thus, an SFI system that satisfies Disjoint Noninterference is guaranteed to not leak data while running within the sandbox.

We formalize NaCl style heavyweight transitions in the technical appendix [Kolosick et al. 2021] and prove that they meet the above secure transition properties. We discuss our proof that zero-cost Wasm meets the above secure transition properties in Section 6.4.

4 ZERO-COST TRANSITION CONDITIONS

Having laid out the security properties required of an SFI transition system, we turn to formally defining the set of zero-cost conditions on sandboxed code such that they sufficiently capture when we may securely elide springboards or trampolines. To this end we define our zero-cost conditions as a safety monitor via the language oSFIasm overlaid on top of SFIasm. oSFIasm extends SFIasm with additional structure and dynamic type checks that ensure the invariants needed for zero-cost transitions are maintained upon returning from library functions, providing both an inductive structure for proofs of security for zero-cost implementations and providing a top-level guarantee

[6] Full definitions are in the technical appendix [Kolosick et al. 2021].

$$\textsc{oCall}\quad \frac{\begin{array}{c}\langle n, \texttt{lib}\rangle = \mathcal{V}_\Phi(e) \qquad n' = k(n) \qquad sp' = \Phi.sp + 1 \\ M' = \Phi.M[sp' \mapsto \Phi.pc + 1] \qquad stack' = [SF] +\!\!+\, \Phi.stack \qquad SF = \text{new-frame}(\Phi, n', sp') \\ \text{typechecks}(\Phi, n', sp') \qquad \Phi' = \Phi[stack := stack', pc := n', sp := sp', M := M']\end{array}}{\Phi(\!|\texttt{call}_k\ e|\!)_{\texttt{lib}} \rightsquigarrow \Phi'}$$

$$\textsc{oRet}\quad \frac{\begin{array}{c}\text{is-ret-addr}(\Phi, \Phi.sp) \qquad \langle n\rangle = \Phi.M(\Phi.sp) \qquad n' = k(n) \\ \text{csr-restored}(\Phi) \qquad \Phi' = \text{pop-frame}(\Phi)\end{array}}{\Phi(\!|\texttt{ret}_k|\!)_{\texttt{lib}} \rightsquigarrow \Phi'[pc := n', sp := \Phi.sp - 1]}$$

$$\textsc{oJmp}\quad \frac{\begin{array}{c}\langle n, \texttt{lib}\rangle = \mathcal{V}_\Phi(e) \qquad n' = k(n) \\ \text{in-same-func}(\Phi, \Phi.pc, n')\end{array}}{\Phi(\!|\texttt{jmp}_k\ e|\!)_{\texttt{lib}} \rightsquigarrow \Phi[pc := n']}$$

$$\textsc{oStore}\quad \frac{\begin{array}{c}\langle n, \texttt{lib}\rangle = \mathcal{V}_\Phi(e) \qquad v = \langle _, p_{e'}\rangle = \mathcal{V}_\Phi(e') \qquad M' = \Phi.M[n' \mapsto v] \\ \text{writeable}(\Phi, n') \qquad n' = k(n) \qquad p_{e'} = \mathsf{app} \Longrightarrow n' \notin H_{\texttt{lib}}\end{array}}{\Phi(\!|\texttt{store}_k\ e := e'|\!)_{\texttt{lib}} \rightsquigarrow \Phi^{++}[M := M']}$$

Fig. 4. oSFIasm Operational Semantics Excerpt

that our integrity and confidentiality properties are maintained. In Section 4.2 we outline the proofs of overlay soundness, showing that oSFIasm captures when a system is zero-cost secure.

Syntax of oSFIasm. Figure 3 shows the extended syntax of oSFIasm. Values (v) are extended with a security label p. Overlay state, written Φ, wraps the state of SFIasm, extending it with two extra pieces of data. First, oSFIasm requires the sandboxed code be organized into functions ($\Phi.funcs$). $\Phi.funcs$ maps each command in the sandboxed library to its parent function. Functions (F) also store the code indices of their commands as the field $F.instrs$, store the entry point ($F.entry$), and track the number of arguments the function expects ($F.type$). This partitioning of sandboxed code into functions is static. Second, the overlay state dynamically tracks a list of overlay stack frames ($\Phi.stack$). These stack frames (SF) are solely logical and inaccessible to instructions. They instead serve as bookkeeping to implement the dynamic type checks of oSFIasm by tracking the base address of each stack frame ($SF.base$), the stack location of the return address ($SF.ret\text{-}addr$), and the values of the callee save registers upon entry to the function ($SF.csr\text{-}vals$). We are concerned with the behavior of the untrusted library, so the logical stack does not finely track application stack frames, but keeps a single large "stack frame" for all nested application stack frames.

When code fails the overlay's dynamic checks it will result in the state `oerror`. Our definition of monitor safety, which will ensure that zero-cost transitions are secure, is then simply that a program does not step to an `oerror`.

4.1 Overlay Monitor

oSFIasm enforces our zero-cost conditions by extending the operational semantics of SFIasm with additional checks in the overlay's small step operational semantics, written $\Phi \rightsquigarrow \Phi'$. Each of these steps is a refinement of the underlying SFIasm step, that is $\Phi.\Psi \rightarrow \Phi'.\Psi$ whenever Φ' is not `oerror`. Figure 4 (with auxiliary definitions shown in Figure 5) shows an excerpt of the checks, which we describe below. Full definitions can be found in the technical appendix [Kolosick et al. 2021]. The checks are similar in nature to the defensive semantics of Besson et al. [2018] though they account for confidentiality and define a more flexible notion of protecting stack frames.

Call. In the overlay, the reduction rule for library `call` instructions (oCall) checks type-safe execution with typechecks, a predicate over the state (Φ), call target ($target$), and stack pointer (sp)

$$\frac{F = \Phi.funcs(target) \quad F.entry = target \quad sp \in S_p \quad [SF] +\!\!+\ _ = \Phi.stack \quad sp \geq SF.ret\text{-}addr + F.type}{\text{typechecks}(\Phi, target, sp)}$$

$$\frac{[SF] +\!\!+\ _ = \Phi.stack \quad ret\text{-}addr = SF.ret\text{-}addr}{\text{is-ret-addr}(\Phi, ret\text{-}addr)}$$

$$\frac{F \in \text{cod}(\Phi.funcs) \quad n, n' \in F.instrs}{\text{in-same-func}(\Phi, n, n')}$$

$$\frac{[SF] +\!\!+\ _ = \Phi.stack \quad \forall(r, n) \in SF.csr\text{-}vals.\ \Phi.R(r) = n}{\text{csr-restored}(\Phi)}$$

$$\frac{[SF] +\!\!+\ _ = \Phi.stack \quad n \in S_p \Longrightarrow n \geq SF.base \wedge n \neq SF.ret\text{-}addr}{\text{writeable}(\Phi, n)}$$

Fig. 5. oSFIasm Semantics Auxiliary Predicates

that checks that (1) the address we are jumping to is the entry instruction of one of the functions, (2) the stack pointer remains within the stack ($sp \in S_p$), and (3) the number of arguments expected by the callee have been pushed to the stack. On top of this, `call` also creates a new logical stack frame recording the base of the new frame, location of the return address, and the current callee-save register values, pushing the new frame onto the overlay stack. To ensure IFC, we require that i has the label `lib` to ensure that control flow is not influenced by confidential values; a similar check is done when jumping within library code, obviating the need for a program counter label. Further, because the overlay captures zero-cost transitions, `gatecall` behaves in the exact same way except for an additional IFC check that the arguments are not influenced by confidential values.

Jmp. Our zero-cost conditions rely on preventing invariants internal to a function from being interfered with by other functions. A key protection enabling this is illustrated by the reduction for `jmp` (oJmp), which enforces that the only inter-function control flow is via `call` and `ret`: the in-same-func predicate checks that the current (n) and target (n') instructions are within the same overlay function. The same check is added to the program counter increment operation, Φ^{++}. These checks ensure that the logical call stack corresponds to the actual control flow of the program, enabling the overlay stack's use in maintaining invariants at the level of function calls.

Store. The reduction rule for `store` (oStore) demonstrates the other key protection enabling function local reasoning, with the check that the address (n) is writeable given the current state of the overlay stack. The predicate writeable guarantees that, if the operation is writing to the stack, then that write must be within the current frame and cannot be the location of the stored return address. This allows reasoning to be localized to each function: they do not need to worry about their callees tampering with their local variables. Protecting the stored return address is crucial for ensuring well-bracketing, which guarantees that each function returns to its caller.

To guarantee IFC, oStore first requires that the pointer have the label `lib`, ensuring that the location we write to is not based on confidential data. Second, the check $p_{i'} = \mathsf{app} \Longrightarrow n' \notin H_{\mathtt{lib}}$ enforces that confidential values cannot be written to the library heap. Similar checks, based on standard IFC techniques, are implemented for all other instructions.

Ret. With control flow checks and memory write checks in place, we guarantee that, when we reach a `ret` instruction, the logical call frame will correspond to the "actual" call frame. `ret` is then responsible for guaranteeing well-bracketing and ensuring callee-save registers are restored. This is handled by two extra conditions on `ret` instructions: is-ret-addr and csr-restored. csr-restored checks that callee-save registers have been properly restored by comparing against the values that were saved in the logical stack frame by `call`. is-ret-addr checks that the value pointed to by the stack pointer (*ret-addr*) corresponds to the location of the return address saved in the logical stack frame. Memory writes were checked to enforce that the return address cannot be overwritten, so this guarantees the function will return to the expected program location.

4.2 Overlay Semantics Enforce Security

The goal of the overlay semantics and our zero-cost conditions is to capture the essential behavior necessary to ensure that individual, well-behaved library functions can be composed together into a sandboxed library call that enforces SFI integrity and confidentiality properties. Thus, library code that is well-behaved under the dynamic overlay type system will behave equivalently to library code with springboard and trampoline wrappers, and therefore well-behaved library code can safely elide those wrappers and their overhead. We prove that the overlay semantics is sound with respect to each of our security properties:

THEOREM 1 (OVERLAY INTEGRITY SOUNDNESS). *If* $\Phi_0 \in \mathit{Program}$, $\Phi_0 \rightsquigarrow^n \Phi_1$, $\Phi_1(\!|_|\!)_{\mathsf{app}}$, *and* $\Phi_1 \rightsquigarrow^* \Phi_2$ *such that* $\Phi_1.\Psi \xrightarrow{wb} \Phi_2.\Psi$ *with* $\pi = \Phi_0.\Psi \rightarrow^n \Phi_1.\Psi$, *then (1)* $\mathcal{CSR}(\pi, \Phi_1.\Psi, \Phi_2.\Psi)$ *and (2)* $\mathcal{RA}(\pi, \Phi_1.\Psi, \Phi_2.\Psi)$.

THEOREM 2 (OVERLAY CONFIDENTIALITY SOUNDNESS). *If* $\Phi_0 \in \mathit{Program}$, $\Phi_1(\!|_|\!)_{\mathtt{lib}}$, $\Phi_3(\!|_|\!)_{\mathsf{app}}$, $\Phi_0.\Psi \rightarrow^* \Phi_1.\Psi \xrightarrow{\mathtt{lib}}{}^n \Phi_2.\Psi \rightarrow \Phi_3.\Psi$, $\Phi_1 \rightsquigarrow^{n+1} \Phi_3$, *and* $\Phi_1 =_{\mathtt{lib}} \Phi'_1$, *then* $\Phi'_1.\Psi \xrightarrow{\mathtt{lib}}{}^n \Phi'_2.\Psi \rightarrow \Phi'_3.\Psi$, $\Phi'_1 \rightsquigarrow^{n+1} \Phi'_3$ $\Phi'_3(\!|_|\!)_{\mathsf{app}}$, $\Phi_3.pc = \Phi'_3.pc$, *and (1)* $\Phi_2(\!|\mathtt{gatecall}_{n'}\ e|\!)$, $\Phi'_2(\!|\mathtt{gatecall}_{n'}\ e|\!)$, *and* $\Phi_3 =_{\mathtt{call}\ n'} \Phi'_3$ *or (2)* $\Phi_2(\!|\mathtt{gateret}|\!)$, $\Phi'_2(\!|\mathtt{gateret}|\!)$, *and* $\Phi_3 =_{\mathtt{ret}} \Phi'_3$.

5 INSTANTIATING ZERO-COST

We describe two isolation systems that securely support zero-cost transitions: they meet the overlay monitor zero-cost conditions. The first (§5.1) is an SFI system using WebAssembly as an IR before compiling to native code using the Lucet toolchain [Bytecode Alliance 2020a]. Here we rely on the language-level invariants of Wasm to satisfy our zero-cost requirements. To ensure that these invariants are maintained, in Section 6 we describe a verifier, VeriZero, that checks that compiled binaries meet the zero-cost conditions. In Section 6.4 we outline our proof that the verifier guarantees that compiled Wasm can safely elide springboards and trampolines.

The second system, `SegmentZero32`, is our novel SFI system combining the x86 segmented memory model for memory isolation with several security-hardening LLVM compiler passes to enforce our zero-cost conditions. While WebAssembly meets the zero-cost conditions, it imposes additional restrictions that lead to unrelated slowdowns. `SegmentZero32` thus serves as a platform for evaluating the potential cost of enforcing the zero-cost conditions directly as well as a proof-of-concept SFI implementation designed using the zero-cost framework.

5.1 WebAssembly

WebAssembly (Wasm) is a low-level bytecode with a sound, static type system. Wasm's abstract state includes global variables and heap memory, which are zero-initialized at start-up. All heap accesses are explicitly bounds checked, meaning that compiled Wasm programs inherently implement heap isolation. Beyond this, Wasm programs enjoy several language-level properties, which ensure compiled binaries satisfying the zero-cost conditions. We describe these below.

Control Flow. There are no arbitrary jump instructions in Wasm, only structured intra-function control flow. Functions may only be entered through a call instruction, and may only be exited by executing a return instruction. Functions also have an associated type; direct calls are type-checked at compile time while indirect calls are subject to a runtime type check. This ensures that compiled Wasm meets our type-directed forward-edge CFI condition.

Protecting the Stack. A Wasm function's type precisely describes the space required to allocate the function's stack frame (including spilled registers). All accesses to local variables and arguments are performed through statically known offsets from the current stack base. It is therefore impossible

for a Wasm operation to access other stack frames or alter the saved return address. This ensures that compiled Wasm meets our local state encapsulation condition, and, in combination with type-checking function calls, guarantees that Wasm's control-flow is well-bracketed. We therefore know that compiled Wasm functions will always execute the register-saving preamble and, upon termination, will execute the register-restoring epilogue. Further, the function body will not alter the values of any registers saved to the stack, thereby ensuring restoration of callee-save registers.

Confidentiality. Wasm code may store values into function-local variables or a function-local "value stack" similar to that of the Java Virtual Machine [jvm 2019]. The Wasm spec requires that compilers initialize function-local variables either with a function argument or with a default value. Further, accesses to the Wasm value stack are governed by a coarse-grained data-flow type system, with explicit annotations at control flow joins. These are used to check at compile-time that an instruction cannot pop a value from the stack unless a corresponding value was pushed earlier in the same function. This guarantees that local variable and value stack accesses can be compiled to register accesses or accesses to a statically-known offset in the stack frame.

When executing a compiled Wasm function without heavyweight transitions, confidential values from prior computations may linger in these spilled registers or parts of the stack. However, the above checks ensure that these locations will only be read if they have been previously overwritten during execution of the same function by a low-confidentiality Wasm library value.

5.2 SegmentZero32

To demonstrate that zero-cost conditions can be applied outside of highly structured languages such as Wasm, we demonstrate their enforcement in our novel SFI system for C code called `SegmentZero32`. As we mention in §2.3, our zero-cost conditions amalgamate a number of individual conditions which separately have well-studied enforcement mechanisms, and so we are able to compose a series of off-the-shelf Clang/LLVM security-hardening passes to form the core of `SegmentZero32`. The memory bounds checks are performed using the x86 segmented memory model [Intel 2020] (Similar to NaCl [Yee et al. 2009], however we use an additional segment to separate the sandboxed heap and stack).

Since `SegmentZero32` directly enforces the structure required for zero-cost transitions on C code (rather than relying on Wasm as an IR), it allows us to investigate the intrinsic cost of enforcing zero-cost (See Section 7.3), without suffering from irrelevant Wasm overheads. We additionally compare `SegmentZero32` against NaCl's 32-bit SFI scheme for the x86 architecture, which we believe is the fastest production-quality SFI toolchain currently available. Below we discuss specific details `SegmentZero32` zero-cost condition enforcement.

Protecting the Stack. We apply the SafeStack [Kuznetsov et al. 2014; The LLVM Foundation 2021b] compiler pass to further split the sandboxed stack into a safe and unsafe stack. The safe stack contains only data that the compiler can statically verify is always accessed safely, e.g., return addresses, spilled registers, and allocations that are only accessed locally using verifiably safe offsets within the function that allocates them.[7] All other stack values are moved to the heap segment. This ensures that pointer manipulation of unsafe stack references cannot be used to corrupt the return address and saved context of the current call. We write a small LLVM pass to add additional support for tracking whether an access must be made through the heap segment or the stack segment, ensuring correct code generation.

These transformations ensure that malicious code cannot programmatically access anything stored in the stack segment, except through offsets statically determined to be safe by the SafeStack

[7]We also use LLVM's stack-heap clash detection (-fstack-clash-protection) to prevent the stack growing into the heap.

```
1  bad_func: [] ⟶ rax
2    push r12
3    ; TRACK: stack[0] = initial r12 value
4    mov r12 ⟵ 1
5    ; TRACK: r12 initialized
6    mov r11 ⟵ r13 + r12
7    ; TRACK: r11 uninitialized
8    mov rdi ⟵ 2
9    ; TRACK: rdi initialized
10   ; ASSERT: good_func arguments initialized
11   call good_func
12   ; TRACK: good_func return value initialized
13   pop r12
14   ; TRACK: r12 = initial r12 value
15   ; ASSERT: callee-save registers restored
16   gateret
```

```
good_func: [rdi] ⟶ rax
  mov rax ⟵ rdi
  ret
```

Fig. 6. Disassembled and lifted WebAssembly functions

pass. This protects the stored callee-save registers and return address, guaranteeing the restoration of callee-save registers and well-bracketing *iff forward control flow is enforced.*

Control Flow. Fortunately, enforcing forward-edge CFI has been widely studied [Burow et al. 2017]. We use a CFI pass as implemented in Clang/LLVM [The LLVM Foundation 2021a; Tice et al. 2014] including flags to dynamically protect indirect function calls, ensuring forward control flow integrity. Further, `SegmentZero32` conservatively bans non-local control flow (e.g. `setjmp/longjmp`) in the C source code. A more permissive approach is possible, but we leave this for future work.

Confidentiality. To guarantee confidentiality we implement a small change in Clang to zero initialize all stack variables.[8] This ensures that scratch registers cannot leak secrets as all sandbox values are semantically written before use. In practice, many of these writes are statically known to be dead and therefore optimised away.

6 VERIFYING COMPILED WEBASSEMBLY

Instead of trusting the Wasm compiler, we build a *zero-cost verifier*, VeriZero, to check that the native, compiled output meets the zero-cost conditions and is thus safe to run without springboards and trampolines. VeriZero is a static x86 assembly analyzer that takes as input potentially untrusted native programs and verifies a series of local properties via abstract interpretation. Together these local properties guarantee that the monitor checks defined in oSFIasm are met; we discuss the proof of soundness in Section 6.4.

VeriZero extends the VeriWasm SFI verifier [Johnson et al. 2021]. Both operate over WebAssembly modules compiled by the Lucet Wasm compiler [Bytecode Alliance 2020a], first disassembling the native x86 code before computing a control-flow graph (CFG) for each function in the binary. The disassembled code is then lifted to a subset of SFIasm, which serves as the first abstract domain in our analysis. Unfortunately, the properties checked by VeriWasm, while sufficient to guarantee SFI security, are insufficient to guarantee zero-cost security. Below we will describe how VeriZero extends VeriWasm to guarantee the stronger zero-cost conditions are met.

[8]We can't use Clang's existing pass for variable initialization [The LLVM Foundation 2018] as it zero initializes data on the unsafe stack leading to poor performance

6.1 The VeriZero Analyzers

VeriZero adds two new analyses to VeriWasm. The first extends VeriWasm's CFI analysis, which only captures coarse grained control-flow (i.e., that all calls target valid sandboxed functions), to also extract type information. Extracting type information from the binary code is possible without any complex type inference because Lucet leaves the type signatures in the compiled output (though we do not need to trust Lucet to get these type signatures correct since VeriZero would catch any deviations at the binary level). For direct calls, VeriZero simply extracts the WebAssembly type stored in the binary. For indirect calls we extend the VeriWasm indirect call analysis to track the type of each indirect call table entry, enabling us to resolve each indirect call to a statically known type. These types correspond to the input registers and stack slots, and the output registers (if any) used by a function. For example, in Figure 6 `bad_func` takes no input and outputs to `rax` and `good_func` takes `rdi` as input and outputs to `rax`.

The second analysis tracks dataflow in local variables, i.e., in registers and stack slots. Continuing with `bad_func` as our example this analysis captures that: in Line 2 stack slot 0 now holds the initial value of `r12`, in Line 4 `r12` holds an initialized (and therefore public) value, in Line 6 `r13` has not been initialized and therefore potentially contains confidential data so `r11` may also contain confidential data, etc. This analysis is used to check confidentiality, callee-save register restoration, local state encapsulation, and is combined with the previous analysis to check type-directed CFI.

6.2 The Dataflow Abstract Domain

To track local variable dataflow, VeriZero uses an abstract domain with three elements: `Uninitialized` which represents an uninitialized, potentially confidential value; `Initialized` which represents an initialized, public value; and `UninitializedCallee`(r) which represents a potentially confidential value which corresponds to the original value of the callee-save register r. The domain forms a meet-semilattice with `Uninitialized` the least element and all other elements incomparable.

From here, analysis is straightforward, with a function's argument registers and stack slots initialized to `Initialized`, each callee-save register r initialized to `UninitializedCallee`(r), and everything else `Uninitialized`. Instructions are interpreted as expected, e.g., `mov` simply copies the abstract value of its source into the target, operations return the meet of their operands, and all constants and reads from the heap are treated as initialized. Across calls we assume that callee-save register conventions are followed (as we will be checking this), preserving the value of all callee-save registers and clearing all other registers' values. We extract the type information from the extended CFI analysis to determine the return register that is initialized after a function call.

6.3 Checking the Zero-Cost Conditions

The above two analyses, along with additional information from VeriWasm's existing analyses enable us to check the zero-cost conditions.

(1) *Callee-save register restoration:* The `UninitializedCallee`(r) value enables straightforward checking that callee-save registers have been restored by checking that, at each `ret` instruction, each callee-save register r has the abstract value `UninitializedCallee`(r).
(2) *Well-bracketed control-flow:* VeriWasm already implements a stack checker that guarantees that all writes to the stack are to local variables, ensuring that the saved return address on the stack cannot be tampered with. Further, it checks that the stack pointer is restored to its original location at the end of every function, ensuring the saved return address is used.
(3) *Type-directed forward-edge CFI:* The dataflow analysis gives us the registers that are initialized when we reach a `call` instruction, enabling us to check that the input arguments of the target

have been initialized. For example, when we reach Line 11 we know that `rdi` has the value `Initialized`. The type-based CFI analysis tells us that `good_func` expects `rdi` as an input, so this call is marked as safe.

(4) *Local state encapsulation:* To ensure SFI security, VeriWasm checks that no writes are below the current stack frame, ensuring that verified Wasm functions cannot tamper with other frames.

(5) *Confidentiality:* We check confidentiality using the information obtained in our dataflow analysis, where the value `Initialized` ensures that a value is initialized with a public, non-confidential value. This enables us to check each of the confidentiality checks encoded in oSFIasm are met: for instance the type-safe forward-edge CFI check described above already ensures each argument is initialized. In Figure 6, the confidentiality checker will flag Line 6 as unsafe because `r13` still has the value `UninitializedCallee(r13)`, which potentially contains confidential information leaked from the application.

6.4 Proving Wasm Secure

We prove that compiled and verified Wasm libraries can safely elide springboards and trampolines while maintaining integrity and confidentiality, by showing that the verified code would not violate the safety monitor. Formally, this amounts to showing that Wasm code verified by VeriZero never reaches an `oerror` state. This allows us to apply Theorem 1 and Theorem 2. It is relatively straightforward (with one exception) to prove that the abstract interpretation as described guarantees the necessary safety conditions.

The crucial exception in the soundness proof is when a function calls to other Wasm functions. We must inductively assume that the called function is safe, i.e., doesn't change any variables in our stack frame, restores callee-save registers, etc. Unfortunately, a naive attempt does not lead to an inductively well-founded argument. Instead, we use the overlay monitor's notion of a well-behaved function to define a step-indexed logical relation (detailed in the technical appendix [Kolosick et al. 2021]) that captures a semantic notion of well-behaved functions (as a relation $\mathcal{F}$), and then lift this to a relation over an entire Wasm library (as a relation $\mathcal{L}$). This gives a basis for an inductively well-founded argument where we can prove that, locally, the abstract interpretation gives that each Wasm function is semantically well-behaved (is in $\mathcal{F}$) and then use this to prove the standard fundamental theorem of a logical relation for a whole Wasm library:

Theorem 3 (Fundamental Theorem for Wasm Libraries). *For any number of steps $n \in \mathbb{N}$ and compiled Wasm library L, $(n, L) \in \mathcal{L}$.*

This theorem states that every function in a compiled Wasm library, when making calls to other Wasm functions or application callbacks, is well-behaved with respect to the zero-cost conditions. The number of steps is a technical detail related to step-indexing. Zero-cost security then follows by adequacy of the logical relation, Theorem 1, and Theorem 2:

Theorem 4 (Adequacy of Wasm Logical Relation). *For any number of steps $n \in \mathbb{N}$, library L such that $(n, L) \in \mathcal{L}$, program $\Phi_0 \in$ Program using L, and $n' \leq n$, if $\Phi_0 \rightsquigarrow^{n'} \Phi'$ then $\Phi' \neq$ `oerror`.*

Details of the logical relation and proofs are in the technical appendix [Kolosick et al. 2021].

7 EVALUATION

We evaluate our zero-cost model by asking four questions:

- **Q1**: What is the cost of a context switch? (§7.1)
- **Q2**: What is end-to-end performance gain of Wasm-based SFI due to zero-cost transitions? (§7.2)
- **Q3**: What is the performance overhead of purpose-built zero-cost SFI enforcement? (§7.3)

► **Q4**: Is the VeriZero verifier effective? (§7.4)

Since our zero-cost condition enforcement does incur some runtime overhead, **Q2** and **Q3** are heavily workload-dependent. The benefit a workload receives from the zero-cost approach will be in direct proportion to the frequency with which it performs domain transitions.

Systems. To investigate the first three questions, we consider two groups of SFI systems. The first group compares a number of different transition models for Wasm-based SFI for 64-bit binaries, built on top of the Lucet compiler [Bytecode Alliance 2020a]. All of these will have identical runtime overhead, meaning that the only variance between them will be due to transition overhead. The `WasmLucet` build uses the original heavyweight springboards and trampolines shipped with the Lucet runtime written in Rust. `WasmHeavy` adopts techniques from NaCl and uses optimized assembly to save and restore application context during transitions. `WasmZero` implements our zero-cost transition system, meaning transitions are simple function calls. To understand the overhead of register saving/restoring and stack switching, we also evaluate a `WasmReg` build which saves/restores registers like `WasmHeavy`, but shares the library and application stack like `WasmZero`.

The second group compares optimized SFI techniques for 32-bit binaries. Wasm-based SFI imposes overheads far beyond what is strictly necessary to enforce our zero-cost conditions, both because of the immaturity of the Lucet compiler in comparison to more established compilers such as Clang, and because Wasm inherently enforces additional restrictions on compiled code (e.g., structured intra-function control flow). We design `SegmentZero32` (§5.2) to enforce only our zero-cost-conditions and nothing more, aiming to benchmark it against the Native Client 32-bit isolation scheme (`NaCl32`) [Yee et al. 2009], arguably the fastest production SFI system available, which requires heavyweight transitions. Both systems make use of memory segmentation, a 32-bit x86-only feature for fast memory isolation. Unfortunately, we cannot make a uniform comparison between `NaCl32`, `SegmentZero32`, and `WasmZero` since Lucet only supports a 64-bit target.

Each group additionally uses unsandboxed, insecure native execution (`Vanilla`) as a baseline. To represent the best possible performance of schemes relying on heavyweight transitions, we also benchmark `IdealHeavy32` and `IdealHeavy64`, ideal hardware isolation schemes, which incur no runtime overhead but require heavyweight transitions. To simulate the performance of these ideal schemes, we simply measure the performance of native code with heavyweight trampolines.

We integrate all of the above SFI schemes into Firefox using the RLBox framework [Narayan et al. 2020]. Since RLBox already provides plugins for the `WasmLucet` and `NaCl32` builds, we only implement the plugins for the remaining system builds.

Benchmarks. We use a micro-benchmark to evaluate the cost of a single transition for our different transition models, using unsandboxed native calls as a baseline (**Q1**).

We answer questions **Q2**–**Q3** by measuring the end-to-end performance of font and image rendering in Firefox, using a sandboxed `libgraphite` and `libjpeg`, respectively. We use these libraries because they have many cross-sandbox transitions, which Narayan et al. [2020] previously observed to affect the overall browser performance. To evaluate the performance of `libgraphite`, we use Kew's benchmark[9], which reflows the text on a page ten times, adjusting the size of the fonts each time to negate the effects of font caches. When calling `libgraphite`, Firefox makes a number of calls into the sandbox roughly proportional to the number of glyphs on the page. We run this benchmark 100 times and report the median execution time below (all values have standard deviations within 1%).

To evaluate the performance of `libjpeg`, we measure the overhead of rendering images of varying complexity and size. Since the work done by the sandboxed `libjpeg`, per call, is proportional to the width of the image — Firefox executes the library in *streaming mode*, one row at a time — we

[9] Available at https://jfkthame.github.io/test/udhr_urd.html

Table 1. Costs of transitions in different isolation models. Zero-cost transitions are shown in **boldface**. `Vanilla` is the performance of an unsandboxed C function call, to serve as a baseline.

Build	Direct call	Indirect call	Callback	Syscall
`Vanilla` (in C)	1ns	56ns	56ns	24ns
`WasmLucet`	—	1137ns	—	—
`WasmHeavy`	120ns	209ns	172ns	192ns
`WasmReg`	120ns	210ns	172ns	192ns
`WasmZero`	**7ns**	**66ns**	**67ns**	**60ns**
`Vanilla` (in C, 32-bit)	1ns	74ns	74ns	37ns
`NaCl32`	—	714ns	373ns	356ns
`SegmentZero32`	**24ns**	**108ns**	**80ns**	**88ns**

consider images of different widths, keeping the image height fixed. This allows us to understand the benefits and limitations of zero-cost transitions, since the proportion of execution time spent context-switching decreases as the image width increases. We do this for three images, of varying complexity: a simple image consisting of a single color (`SimpleImage`), a stock image from the Image Compression benchmark suite[10] (`StockImage`), and an image of random pixels (`RandomImage`). We render each image 500 times and report the median time (standard deviations are all under 1%).

Finally, we use SPEC CPU® 2006 to partly evaluate the sandboxing overhead of our purpose-built `SegmentZero32` SFI system (**Q3**), and to measure VeriZero's verification speed (**Q4**).

Machine and Software Setup. We run all but the verification benchmarks on an Intel® Core™ i7-6700K machine with four 4GHz cores, 64GB RAM, running Ubuntu 20.04.1 LTS (kernel version 5.4.0-58). We run benchmarks with a shielded isolated cpuset [Tsariounov 2021] consisting of one core with hyperthreading disabled and the clock frequency pinned to 2.2GHz. We generate Wasm sandboxed code in two steps: First, we compile C/C++ to Wasm using Clang-11, and then compile Wasm to native code using the fork of the Lucet used by RLBox (snapshot from Dec 9, 2020). We generate NaCl sandboxed code using a modified version of Clang-4. We compile all other C/C++ source code, including `SegmentZero32` sandboxed code and benchmarks using Clang-11. We implement our Firefox benchmarks on top of Firefox Nightly (from August 22, 2020).

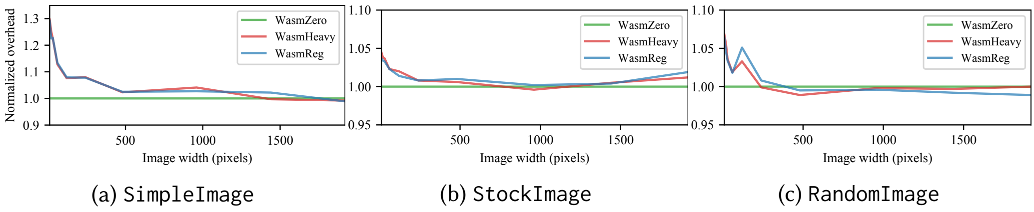
Summary of Results. We find that the performance of Wasm-based isolation can be significantly improved by adopting zero-cost transitions, but that Lucet-compiled WebAssembly's runtime overhead means that it does not outperform more optimised isolation schemes in end-to-end tests. The low performance overhead of `SegmentZero32` demonstrates that these runtime overheads are not inherent to the zero-cost approach, and that an optimised zero-cost SFI system can significantly outperform more traditional schemes, especially for workloads with a large number of transitions. Finally, we find that we can efficiently check zero-cost conditions at the binary level, for Lucet compiled code, with no false positives.

7.1 The Cost of Transitions

We measure the cost of different cross-domain transitions — direct and indirect calls into the sandbox, callbacks from the sandbox, and syscall invocations from the sandbox — for the different system builds described above. To expose overheads fully, we choose extremely fast payloads—either a function that just adds two numbers or the `gettimeofday` syscall, which relies on Linux's vDSO to avoid CPU ring changes. The results are shown in Figure 1. All numbers are averages of one million repetitions, and repeated runs have negligible standard deviation.[11]

[10] Online: https://imagecompression.info/test_images/. Visited Dec 9, 2020.

[11] Lucet and NaCl don't support direct sandbox calls; Lucet further does not support custom callbacks or syscall invocations.



(a) `SimpleImage` (b) `StockImage` (c) `RandomImage`

Fig. 7. Performance of different Wasm transitions on rendering of (a) a simple image with one color, (b) a stock image, and (c) a complex image with random pixels, normalized to `WasmZero`. `WasmZero` transitions outperform other transitions. The difference diminishes with width, but narrower images are more common on the web.

We make several observations. First, among Wasm-based SFI schemes, zero-cost transitions (`WasmZero`) are significantly faster than even optimized heavyweight transitions (`WasmHeavy`). Lucet's existing indirect calls written in Rust (`WasmLucet`) are significantly slower than both. Second, the cost of stack switching (the difference of `WasmHeavy` and `WasmReg`) is surprisingly negligible. Third, the performance of `Vanilla` and `WasmZero` should be identical but is not. This is *not* because our transitions have a hidden cost. Rather, it's because we are comparing code produced by two different compilers: `Vanilla` is native code produced by Clang, while `WasmZero` is code produced by Lucet, and Lucet's code generation is not yet highly optimized [Hansen 2019]. For example, in the benchmark that adds two numbers, Clang eliminates the function prologue and epilogue that save and restore the frame pointer, while Lucet does not. We observe similar trends for hardware-based isolation. For example, we find that `SegmentZero32` transitions are much faster than `IdealHeavy32` and `NaCl32` transitions and only 23ns slower than `Vanilla` for direct calls. Finally, we observe that `SegmentZero32` is slower than `WasmZero`: hardware isolation schemes like `SegmentZero32` and `NaCl32` execute instructions to enable or disable the hardware based memory isolation in their transitions.

7.2 End-to-End Performance Improvements of Zero-Cost Transitions for Wasm

We evaluate the end-to-end performance impact of the different transition models on Wasm-sandboxed font and image rendering as used in Firefox (see §7).

Font Rendering. We report the performance of `libgraphite` isolated with Wasm-based schemes on Kew's benchmark below:

	WasmLucet	**WasmHeavy**	**WasmReg**	**WasmZero**	**Vanilla**	**IdealHeavy64**
Font render	8173ms	2246ms	2230ms	2032ms	1116ms	1563ms

As expected, Wasm with zero-cost transitions (`WasmZero`) outperforms the other Wasm-based SFI transition models. Compared to `WasmZero`, Lucet's existing transitions slow down rendering by over 4×.[12] But, even the optimized heavyweight transitions (`WasmHeavy`) impose a 10% performance tax. This overhead is due to register saving/restoring; stack switching only accounts for 0.8% overhead.

While these results show that existing Wasm-based isolation schemes can benefit from switching to zero-cost transitions — and indeed the speed-up due to zero-cost transitions allowed Mozilla to ship the Wasm-sandboxed `libgraphite` — they also show that Lucet-compiled Wasm is slow (~80% slower than Vanilla). This, unfortunately, means that the transition cost savings alone are not enough to beat `IdealHeavy64`, even for a workload with many transitions. To compete with

[12]This overhead is smaller than the 8× overhead reported by Narayan et al. [2020]; we attribute this difference to the different compilers — we use a more recent, and faster, version of Lucet.

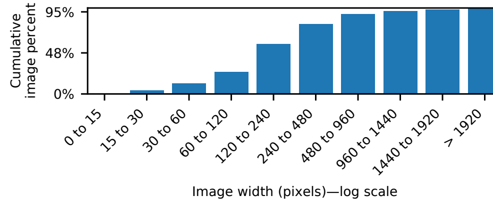


Fig. 8. Cumulative distribution of image widths on the landing pages of the Alexa top 500 websites. Over 80% of the images have widths under 480 pixels. Narrower images have a higher transition rate, and thus higher relative overheads when using expensive transitions.

this ideal SFI scheme with heavyweight transitions, we would need to reduce the runtime overhead to ∼40%. Jangda et al. [2019] report the average runtime overhead of Mozilla SpiderMonkey JIT-compiled WebAssembly compared to native as ∼45% in a different set of benchmarks, while noting many correctable inefficiencies in the generated assembly code, suggesting that there is a lot of room for Lucet to be further optimised.

Image Rendering. Figure 7 report the overhead of Wasm-based sandboxing on image rendering, normalized to `WasmZero` to highlight the relative overheads of different transitions as compared to our zero-cost transitions. We report results of `WasmLucet` separately, in the technical appendix [Kolosick et al. 2021] because the rendering times are up to 9.2× longer than the other builds. Here, we instead focus on evaluating the overheads of optimized heavy transitions.

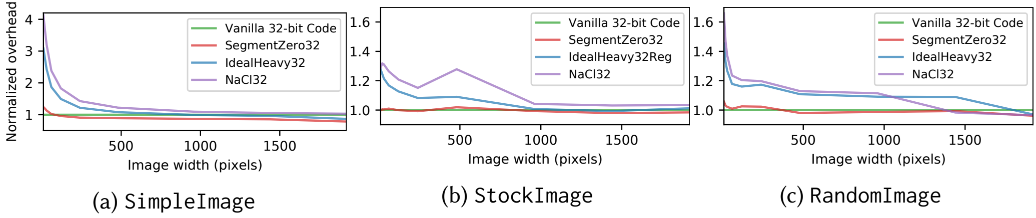
As expected, `WasmZero` significantly outperforms other transitions when images are narrower and simpler. On `SimpleImage`, `WasmHeavy` and `WasmLucet` can take as much as 29.7% and 9.2× longer to render the image as with `WasmZero` transitions. However, this performance gap diminishes as image width increases (and the relative cost of context switching decreases). For `StockImage` and `RandomImage`, the `WasmHeavy` trends are similar, but the rendering time differences start at about 4.5%. Lucet's existing transitions (`WasmLucet`) are still significantly slower than zero-cost transitions (`WasmZero`) even on wide images.

Though the differences between the transitions are smaller as the image width increases, many images on the Web are narrow. Figure 8 shows the distribution of images on the landing pages of the Alexa top 500 websites. Of the 10.6K images, 8.6K (over 80%) have widths between 1 and 480 pixels, a range in which zero-cost transitions noticeably outperform the other kinds of transitions.

Like font rendering, we measure the target runtime overhead Lucet should achieve to beat `IdealHeavy64` end-to-end for rendering images. We report our results in the technical appendix [Kolosick et al. 2021]. For the small simple image, we observe this to be 94% — this is approximately the overhead of Lucet that we see already today. For the small stock image, we observe this to be 15% — this is much smaller than the overhead of Lucet today, but lower overheads have been demonstrated on some benchmarks by the prototype Wasm compiler of Gadepalli et al. [2020].

7.3 Performance Overhead of Purpose-Built Zero-Cost SFI Enforcement

In this section, we measure the performance overhead of `SegmentZero32` with zero-cost transitions. We compare `SegmentZero32` with NaCl (`NaCl32`) and `IdealHeavy32`— a hypothetical SFI scheme with no isolation enforcement overhead, both of which rely on heavyweight transitions. We measure the overhead of these systems on the standard SPEC CPU® 2006 benchmark suite, and the `libgraphite` and `libjpeg` font and image rendering benchmarks. Since both `SegmentZero32` and `NaCl32` use segmentation which is supported only in 32-bit mode, we implement these three isolation builds in 32-bit mode and compare it to native 32-bit unsandboxed code. We describe these benchmarks next.



(a) `SimpleImage` (b) `StockImage` (c) `RandomImage`

Fig. 9. Performance of image rendering with libjpeg sandboxed with `SegmentZero32` and `NaCl32` and `IdealHeavy32`. Times are relative to unsandboxed code. `NaCl32` and `IdealHeavy32` relative overheads are as high as 312% and 208% respectively, while `SegmentZero32` relative overheads do not exceed 24%.

Table 2. Overheads compared to native code on SPEC CPU® 2006 (nc), for `NaCl32` and `SegmentZero32`.

System	400.perlbench	401.bzip2	403.gcc	429.mcf	445.gobmk	456.hmmer	458.sjeng	462.libquantum	464.h264ref	471.omnetpp	473.astar	483.xalancbmk
NaCl32	—	—	1.10×	—	1.27×	0.97×	1.20×	1.06×	1.34×	1.06×	1.31×	—
SegmentZero32	1.20×	1.08×	—	1.04×	1.25×	0.82×	1.16×	1.02×	1.01×	1.01×	1.10×	1.05×

SPEC. We report the impact of sandboxing on SPEC CPU® 2006 in Figure 2. Several benchmarks are not compatible with `NaCl32`; augmenting `NaCl32` runtime and libraries to fix such compatibility issues (e.g., as done in [Yee et al. 2009] for SPEC2000) is beyond the scope of this paper. The gcc benchmark, on the other hand, is not compatible with `SegmentZero32`— gcc fails (at runtime) because the CFI used by `SegmentZero32` — Clang's CFI — incorrectly computes a target CFI label. Clang's CFI implementation is more precise than necessary for our zero-cost conditions; as with `NaCl32`, we leave the implementation of a coarse-grain and more permissive CFI to future work. On the overlapping benchmarks, `SegmentZero32`'s overhead is comparable to `NaCl32`'s.

Font Rendering. The impact of these isolation schemes on font rendering is shown below:

	Vanilla (32-bit)	IdealHeavy32	NaCl32	SegmentZero32
Font render	1441ms	2399ms	2769ms	1765ms

We observe that `NaCl32` and `IdealHeavy32` impose an overhead of 92% and 66% respectively. In contrast, `SegmentZero32` has a smaller overhead (22.5%) as it does not have to save and restore registers or switch stacks. We attribute the overhead of `SegmentZero32` (over Vanilla) to three factors: (1) changing segments to enable/disable isolation during function calls, (2) using indirect function calls for cross-domain calls (a choice that simplifies engineering but is not fundamental), and (3) the structure imposed by our zero-cost condition enforcement.

Image Rendering. We report the impact of sandboxing on image rendering in Figure 9. For narrow images (10 pixel width), `SegmentZero32` overheads relative to the native unsandboxed code are 24%, 1%, and 6.5% for `SimpleImage`, `StockImage` and `RandomImage`, respectively. These overheads are lower than the corresponding overheads for `NaCl32` (312%, 29%, and 66%) as well as `IdealHeavy32` (208%, 28% and 45%). As in the Wasm measurements, these overheads shrink as image width increases and the complexity of the image increases (e.g., the overheads for images wider than 480 pixels are negligible).

7.4 Effectiveness of the VeriZero Verifier

We evaluate VeriZero's effectiveness by using it to (1) verify 13 binaries — five third-party libraries shipping (or about to ship) with Firefox compiled across 3 binaries, and 10 binaries from the SPEC CPU® 2006 benchmarks — and (2) find nine manually introduced bugs, inspired by real calling convention bugs in previous SFI toolchains [Nacl Issue 2919 2012; Nacl Issue 775 2010; Rydgard

2020]. We measure VeriZero's performance verifying the aforementioned 13 binaries. Finally, we stress test VeriZero by running it on random binaries generated by Csmith [Yang et al. 2011].

Experimental Setup. We run all VeriZero experiments on a 2.1GHz Intel® Xeon® Platinum 8160 machine with 96 cores and 1 TB of RAM running Arch Linux 5.11.12. All experiments run on a single core and use no more than 2GB of RAM. We compile the SPEC binaries used using the Lucet toolkit used in Section 7.2. We verify three of the Firefox libraries from Firefox Nightly; we compile the other two from the patches that are in the process of being integrated into Firefox.

Effectiveness and Performance Results. VeriZero successfully verifies the 13 Firefox and SPEC CPU® 2006 binaries. These binaries vary in size from 150 functions (the `lbm` benchmark from SPEC CPU® 2006) to 4094 functions (the binary consisting of the Firefox Nightly libraries `libogg`, `libgraphite`, and `hunspell`). It took VeriZero between 1.77 seconds and 19.28 seconds to verify these binaries, with an average of 9.2 seconds and median of 5.93 seconds. VeriZero's performance is on par with the original VeriWasm's performance: on the 10 SPEC CPU® 2006 benchmarks evaluated in the VeriWasm paper [Johnson et al. 2021] VeriZero is slightly (15%) faster, despite checking zero-cost conditions in addition to all of VeriWasm's original checks. This is due to various engineering improvements that were made to VeriWasm in the course of developing VeriZero.

VeriZero also successfully found bugs injected into nine binaries. These bugs tested all the zero-cost properties that VeriZero was designed to check, and when possible they were based on real bugs (like those in Cranelift [Rydgard 2020]). VeriZero successfully detected all nine of these bugs, giving us confidence that VeriZero is capable of finding violations of the zero-cost conditions.

Fuzzing Results. We fuzzed VeriZero to both search for potential bugs in Lucet, as well as to ensure VeriZero does not incorrectly declare safe programs unsafe. The fuzzing pipeline works in four stages: first, we use Csmith [Yang et al. 2011] to generate random C and C++ programs, next we use Clang to compile the generated C/C++ program to WebAssembly, followed by compiling the Wasm file to native code using Lucet, and finally we verify the generated binary with VeriZero. As these programs were compiled by Lucet, we expect them to adhere to the zero-cost conditions, and any binaries flagged by VeriZero are either bugs in Lucet or are spurious errors in VeriZero.

While we did not find bugs in Lucet, fuzzing did find cases where VeriZero triggered spurious errors. After fixing these errors, we verified 100,000 randomly generated programs with no false positives.

8 LIMITATIONS

Our Wasm SFI scheme is capable of sandboxing any C/C++-like language (with arbitrary intra-function control flow, arbitrary type casting, arbitrary pointer aliasing, arithmetic etc.) that can compile to Wasm, so long as it does not use features which Wasm must emulate through JavaScript[13] — most prominently C++-style exceptions, `setjmp/longjmp`, and multithreading. These limitations are not inherent to our zero-cost conditions, and Wasm is in the process of being extended with support for all of the above features [Watt et al. 2019; WebAssembly Community Group 2021].

Our `SegmentZero32` scheme is built as a proof-of-concept, using mostly existing LLVM passes to sandbox C programs compiled to 32-bit x86, as an approach to understanding the overhead of zero-cost conditions on native code. As such, our `SegmentZero32` implementation does not support, for instance, `setjmp/longjmp` or multithreading (similar to Wasm). It also does not support user-defined variadic function arguments or position independent code. However, these limitations are not fundamental. For example, variadic function arguments could be supported by extending the SafeStack pass to move the variadic argument buffer into the unsafe stack, and position independent code could be supported through minor generalisations of existing compiler primitives.

[13] See https://emscripten.org/

Both Wasm and SegmentZero32 rely on a type-directed forward-edge CFI which requires us to statically infer a limited amount of information about arguments to functions.[14] This information includes the number of arguments, their width, and the calling convention. In practice, this information is readily available as part of compilation and does not require any complex control flow inference (unlike more precise fine grain CFI schemes), so this is only a limitation when analyzing certain binary programs. Like most SFI schemes, both Wasm and our SegmentZero32 do not currently support JIT code compilation within the isolated component; adding this would require engineering work, but can be done following the approaches of [Ansel et al. 2011; Vahldiek-Oberwagner et al. 2019]. Finally, side channels are out of scope for this paper.

9 RELATED WORK

A considerable amount of research has gone into efficient implementations of memory isolation and CFI techniques to provide SFI across many platforms [Adl-Tabatabai et al. 1996; Bittau et al. 2008; Bytecode Alliance 2020a; Castro et al. 2009; Chen et al. 2016; Ford and Cox 2008; Goonasekera et al. 2015; Herder et al. 2009; Litton et al. 2016; Lucco et al. 1995; McCamant and Morrisett 2006; Niu and Tan 2014; Payer and Gross 2011; Sehr et al. 2010; Seltzer et al. 1996; Siefers et al. 2010; Tan et al. 2017]. However, these systems either implement or require the user to implement heavyweight springboards and trampolines to guarantee security.

SFI Systems. Wahbe et al. [1993] suggest two ways to optimize transitions: (1) partitioning the registers used by the application and the sandboxed component and (2) performing link time optimizations (LTO) that conservatively eliminates register saves that are never used in the entire sandboxed component (not just the callee). Register partitioning would cause slowdowns due to increased spilling. Native Client [Yee et al. 2009] optimized transitions by clearing and saving contexts using machine specific mechanisms to, e.g., clear floating point state and SIMD registers in bulk. However, we show (§7) that, even with those optimizations, the software model imposes significant transition overheads. While CPU makers continue to add optimized context switching instructions, such instructions do not yet eliminate all overhead.

Zeng et al. [2011] combine an SFI scheme with a rich CFI scheme enforcing structure on executing code. While a similar approach, their goal is to safely perform optimizations to elide SFI and CFI bounds checks, and they do not impose sufficient structure to enforce well-bracketing, a necessary property for zero-cost transitions. XFI [Erlingsson et al. 2006] also combines an SFI scheme with a rich CFI scheme and adopts a safe stack model. While meeting many of the zero-cost conditions, it does not prevent reading uninitialized scratch registers and therefore cannot ensure confidentiality without heavyweight springboards that clear scratch registers. They also do not specify the CFG granularity, so it is not clear if is strong enough to satisfy the zero-cost type-safe CFI requirement.

WebAssembly Based Isolation. WasmBoxC [Zakai 2020] sandboxes C code by compiling to Wasm followed by (de)compiling back to C, ensuring that the sandboxed code will inherit isolation properties from Wasm. The sandboxed library code can be safely linked with C applications, enabling a form of zero-cost transition. The zero-cost Wasm SFI system described by this paper was designed and released prior to and independently of WasmBoxC, as the creators of WasmBoxC acknowledge (citation elided for DBR). Moreover, we believe that the theory developed in this paper provides a foundation for analyzing and proving the security of WasmBoxC though such analysis would need to account for possible undefined behavior introduced in compiling to C.

Sledge [Gadepalli et al. 2020] describes a Wasm runtime for edge computing, that relies on Wasm properties to enable efficient isolation of serverless components. However, Sledge focuses on

[14]We do not need to infer any information about the heap or unsafe stack. Variadic functions, for example, can pass a dynamic number of arguments on the unsafe stack.

function scheduling including preempting running Wasm programs, so its needs for context saving differ from library sandboxing as contexts must be saved even in the middle of function calls.

SFI Verification. Previous work on SFI (e.g., [Erlingsson et al. 2006; Johnson et al. 2021; McCamant and Morrisett 2006; Yee et al. 2009]) uses a *verifier* or a theorem prover [Kroll et al. 2014; Zhao et al. 2011] to validate the relevant SFI properties of compiled sandbox code. However, unlike VeriZero, none of these verifiers establish sufficient properties for zero-cost transitions.

Hardware Based Isolation. Hardware features such as memory protection keys [Hedayati et al. 2019; Vahldiek-Oberwagner et al. 2019], extended page tables [Qiang et al. 2017], virtualization instructions [Belay et al. 2012; Qiang et al. 2017], or even dedicated hardware designs [Schrammel et al. 2020] can be used to speed up memory isolation. These works focus on the efficiency of memory isolation as well as switching between protected memory domains; however these approaches also use a single memory region that contain both the stack and heap making them incompatible with zero-cost conditions, i.e. they require heavyweight transitions. `IdealHeavy32` and `IdealHeavy64` in Section 7 studies an idealized version of such a scheme.

Capabilities. Karger [1989] and Skorstengaard et al. [2019] look at protecting interacting components on systems that provide hardware-enforced capabilities. Karger [1989] specifically looks at how register saving and restoration can be optimized based on different levels of trust between components, however their analysis does not offer formal security guarantees. Skorstengaard et al. [2019] investigate a calling-convention based on capabilities (à la CHERI [Watson et al. 2015]) that allow safe sharing of a stack between distrusting components. Their definition of well-bracketed control flow and local state encapsulation via an overlay inspired our work, and our logical relation is also based on their work. However, their technique does not yet ensure an equivalent notion to our confidentiality property, and further is tied to machine support for hardware capabilities.

Type Safety for Isolation. There has also been work on using strongly-typed languages to provide similar security benefits. SingularityOS [Aiken et al. 2006; Fähndrich et al. 2006; Hunt and Larus 2007], explored using Sing# to build an OS with cheap transitions between mutually untrusting processes. Unlike the work on SFI techniques that zero-cost transitions extend, tools like SingularityOS require engineering effort to rewrite unsafe components in new safe languages.

At a lower level, Typed Assembly Language (TAL) [Morrisett et al. 1999a, 2002, 1999b] is a type-safe compilation target for high-level type-safe languages. Its type system enables proofs that assembly programs follow calling conventions, and enables an elegant definition of stack safety through polymorphism. Unfortunately, SFI is designed with unsafe code in mind, so cannot generally be compiled to meet TAL's static checks. To handle this our zero-cost and security conditions instead capture the *behavior* that TAL's type system is designed to ensure.

ACKNOWLEDGMENTS

We thank the reviewers for their suggestions and insightful comments. Many thanks to Bobby Holley, Mike Hommey, Chris Fallin, Tom Ritter, Till Schneidereit, Andy Wortman, and Alon Zakai for fruitful discussions. We thank Chris Fallin, Pat Hickey, and Andy Wortman for working with us to integrate VeriZero into the Lucet compiler. This work was supported in part by gifts from Cisco, Fastly, Google, and Intel; by the NSF under Grant Number CNS-1514435, CNS-2120642, CCF-1918573, CAREER CNS-2048262; and, by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Conrad Watt was supported by the EPSRC grant REMS: *Rigorous Engineering for Mainstream Systems* (EP/K008528/1), a Google PhD Fellowship in Programming Technology and Software Engineering, and a Research Fellowship from Peterhouse, University of Cambridge.

REFERENCES

2019. *Java Platform, Standard Edition: Java Virtual Machine Guide.* Technical Report. https://docs.oracle.com/en/java/javase/13/vm/java-virtual-machine-guide.pdf

Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. 1996. Efficient and language-independent mobile programs. In *Proceedings of the ACM SIGPLAN'96 Conference on Programming Language Design and Implementation (PLDI), Philadephia, Pennsylvania, USA, May 21-24, 1996.* ACM. https://doi.org/10.1145/231379.231402

Mark Aiken, Manuel Fähndrich, Chris Hawblitzel, Galen Hunt, and James Larus. 2006. Deconstructing process isolation. In *Proceedings of the 2006 workshop on Memory System Performance and Correctness, San Jose, California, USA, October 11, 2006.* ACM. https://doi.org/10.1145/1178597.1178599

Fritz Alder, Jo Van Bulck, David Oswald, and Frank Piessens. 2020. Faulty Point Unit: ABI Poisoning Attacks on Intel SGX. In *ACSAC '20: Annual Computer Security Applications Conference, Virtual Event / Austin, TX, USA, 7-11 December, 2020.* ACM. https://doi.org/10.1145/3427228.3427270

Jason Ansel, Petr Marchenko, Úlfar Erlingsson, Elijah Taylor, Brad Chen, Derek L. Schuff, David Sehr, Cliff Biffle, and Bennet Yee. 2011. Language-independent sandboxing of just-in-time compilation and self-modifying code. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011.* ACM. https://doi.org/10.1145/1993498.1993540

Alexandre Bartel and John Doe. 2018. Twenty years of escaping the Java sandbox. In *Phrack.*

Adam Belay, Andrea Bittau, Ali Mashtizadeh, David Terei, David Mazières, and Christos Kozyrakis. 2012. Dune: Safe User-Level Access to Privileged CPU Features. In *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation* (Hollywood, CA, USA) *(OSDI'12).* USENIX Association, USA, 335–348.

Frédéric Besson, Sandrine Blazy, Alexandre Dang, Thomas Jensen, and Pierre Wilke. 2019. Compiling sandboxes: Formally verified software fault isolation. In *Programming Languages and Systems - 28th European Symposium on Programming, ESOP 2019, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2019, Prague, Czech Republic, April 6-11, 2019, Proceedings (Lecture Notes in Computer Science, Vol. 11423).* Springer. https://doi.org/10.1007/978-3-030-17184-1_18

Frédéric Besson, Thomas Jensen, and Julien Lepiller. 2018. Modular software fault isolation as abstract interpretation. In *Static Analysis - 25th International Symposium, SAS 2018, Freiburg, Germany, August 29-31, 2018, Proceedings (Lecture Notes in Computer Science, Vol. 11002).* Springer. https://doi.org/10.1007/978-3-319-99725-4_12

Andrea Bittau, Petr Marchenko, Mark Handley, and Brad Karp. 2008. Wedge: Splitting Applications into Reduced-Privilege Compartments. In *5th USENIX Symposium on Networked Systems Design & Implementation, NSDI 2008, April 16-18, 2008, San Francisco, CA, USA, Proceedings*, Jon Crowcroft and Michael Dahlin (Eds.). USENIX Association, 309–322. http://www.usenix.org/events/nsdi08/tech/full_papers/bittau/bittau.pdf

Jay Bosamiya, Benjamin Lim, and Bryan Parno. 2020. WebAssembly as an Intermediate Language for Provably-Safe Software Sandboxing. PriSC.

Nathan Burow, Scott A. Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. 2017. Control-Flow Integrity: Precision, Security, and Performance. *Comput. Surveys* 50 (April 2017), 16:1–16:33. https://doi.org/10.1145/3054924

Bytecode Alliance. 2020a. *Lucet.* https://github.com/bytecodealliance/lucet

Bytecode Alliance. 2020b. *WebAssembly Micro Runtime.* https://github.com/bytecodealliance/wasm-micro-runtime

Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. 2009. Fast byte-granularity software fault isolation. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009.* ACM. https://doi.org/10.1145/1629575.1629581

Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu. 2016. Shreds: Fine-Grained Execution Units with Private Memory. In *2016 IEEE Symposium on Security and Privacy (SP).* 56–71. https://doi.org/10.1109/SP.2016.12

Chromium Team. 2020. Memory safety. https://www.chromium.org/Home/chromium-security/memory-safety.

Úlfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. 2006. XFI: Software guards for system address spaces. In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA.* USENIX Association, 75–88.

Manuel Fähndrich, Mark Aiken, Chris Hawblitzel, Orion Hodson, Galen Hunt, James R Larus, and Steven Levi. 2006. Language support for fast and reliable message-based communication in Singularity OS. In *Proceedings of the 2006 EuroSys Conference, Leuven, Belgium, April 18-21, 2006.* ACM. https://doi.org/10.1145/1217935.1217953

Bryan Ford. 2005. VXA: A Virtual Architecture for Durable Compressed Archives.. In *FAST*, Vol. 5.

Bryan Ford and Russ Cox. 2008. Vx32: Lightweight User-level Sandboxing on the x86. In *2008 USENIX Annual Technical Conference, Boston, MA, USA, June 22-27, 2008. Proceedings.* USENIX Association.

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

Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. 2020. Sledge: a Serverless-first, Light-weight Wasm Runtime for the Edge. In *Middleware '20: 21st International Middleware Conference, Delft, The Netherlands, December 7-11, 2020*. ACM. https://doi.org/10.1145/3423211.3425680

Google Project Zero 2021. Introducing the In-the-Wild Series. https://googleprojectzero.blogspot.com/2021/01/introducing-in-wild-series.html.

Nuwan Goonasekera, William Caelli, and Colin Fidge. 2015. LibVM: an Architecture for Shared Library Sandboxing. 45, 12 (2015), 1597–1617. https://doi.org/10.1002/spe.2294

Matthias Grimmer, Roland Schatz, Chris Seaton, Thomas Würthinger, and Hanspeter Mössenböck. 2015. Memory-safe Execution of C on a Java VM. In *Workshop on Programming Languages and Analysis for Security (PLAS)*.

Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and Jf Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain). Association for Computing Machinery, 185–200. https://doi.org/10.1145/3062341.3062363

Lars T Hansen. 2019. Cranelift: Performance parity with Baldr on x86-64. https://bugzilla.mozilla.org/show_bug.cgi?id=1539399.

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

Jorrit N Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S Tanenbaum. 2009. Fault isolation for device drivers. In *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks, DSN 2009, Estoril, Lisbon, Portugal, June 29 - July 2, 2009*. IEEE Computer Society. https://doi.org/10.1109/DSN.2009.5270357

Galen C Hunt and James R Larus. 2007. Singularity: rethinking the software stack. *SIGOPS Operating Systems Review* 41, 2 (2007).

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

Abhinav Jangda, Bobby Powers, Emery D Berger, and Arjun Guha. 2019. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *2019 USENIX Annual Technical Conference, USENIX ATC 2019, Renton, WA, USA, July 10-12, 2019*. USENIX Association.

Evan Johnson. 2021. *Update VeriWasm version.* https://github.com/bytecodealliance/lucet/pull/684

Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. 2021. Доверя́й, но проверя́й: SFI safety for native-compiled Wasm. In *Network and Distributed System Security Symposium (NDSS)*. Internet Society.

Paul A. Karger. 1989. Using Registers to Optimize Cross-Domain Call Performance. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems* (Boston, Massachusetts, USA) *(ASPLOS III)*. Association for Computing Machinery, New York, NY, USA, 194–204. https://doi.org/10.1145/70082.68201

Matthew Kolosick, Shravan Narayan, Evan Johnson, Conrad Watt, Michael LeMay, Deepak Garg, Ranjit Jhala, and Deian Stefan. 2021. Isolation Without Taxation: Near Zero Cost Transitions for SFI. arXiv:2105.00033 [cs.CR]

Joshua A Kroll, Gordon Stewart, and Andrew W Appel. 2014. Portable software fault isolation. In *2014 IEEE 27th Computer Security Foundations Symposium*. IEEE, 18–32. https://doi.org/10.1109/CSF.2014.10

Volodymyr Kuznetsov, Laszlo Szekeres, Mathias Payer, George Candea, R. Sekar, and Dawn Song. 2014. Code-Pointer Integrity. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*. 147–163. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov

James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation (OSDI'16)*. USENIX Association, 49–64.

H.J. Lu, Michael Matz, Milind Girkar, Jan Hubička, Andreas Jaeger, and Mark Mitchell. 2018. *System V Application Binary InterfaceAMD64 Architecture Processor Supplement(With LP64 and ILP32 Programming Models).* Technical Report. https://software.intel.com/content/dam/develop/external/us/en/documents/intro-to-intel-avx-183287.pdf

Steve Lucco, Oliver Sharp, and Robert Wahbe. 1995. Omniware: A universal substrate for web programming. In *WWW*.

Sergio Maffeis, John C Mitchell, and Ankur Taly. 2010. Object capabilities and isolation of untrusted web applications. In *31st IEEE Symposium on Security and Privacy, S&P 2010, 16-19 May 2010, Berleley/Oakland, California, USA*. IEEE Computer Society, 125–140. https://doi.org/10.1109/SP.2010.16

A.A. Matos and G. Boudol. 2005. On Declassification and the Non-Disclosure Policy. In *18th IEEE Computer Security Foundations Workshop (CSFW'05)*. 226–240. https://doi.org/10.1109/CSFW.2005.21

Stephen McCamant and Greg Morrisett. 2006. Evaluating SFI for a CISC Architecture. In *Proceedings of the 15th USENIX Security Symposium, Vancouver, BC, Canada, July 31 - August 4, 2006*. USENIX Association.

Tyler McMullen. 2020. Lucet: A Compiler and Runtime for High-Concurrency Low-Latency Sandboxing. In *PriSC*.

Kathleen Metrick, Jared Semrau, and Shambavi Sadayappan. 2020. Think Fast: Time Between Disclosure, Patch Release and Vulnerability Exploitation – Intelligence for Vulnerability Management, Part Two. https://www.fireeye.com/blog/threat-research/2020/04/time-between-disclosure-patch-release-and-vulnerability-exploitation.html.

Adrian Mettler, David A Wagner, and Tyler Close. 2010. Joe-E: A Security-Oriented Subset of Java.. In *Network and Distributed System Security Symposium (NDSS)*.

Matt Miller. 2019. Trends, challenges, and strategic shifts in the software vulnerability mitigation landscape. BlueHat.

M.S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. 2008. Caja: Safe active content in sanitized JavaScript. http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf.

Greg Morrisett, Karl Crary, Neal Glew, Dan Grossman, Richard Samuels, Frederick Smith, David Walker, Stephanie Weirich, and Steve Zdancewic. 1999a. TALx86: A Realistic Typed Assembly Language. *ACM SIGPLAN Workshop on Compiler Support for System Software* (1999), 25–35.

Greg Morrisett, Karl Crary, Neal Glew, and David Walker. 2002. Stack-Based Typed Assembly Language. *Journal of Functional Programming* 12 (Jan. 2002), 43–88. https://doi.org/10.1017/S0956796801004178 Publisher: Cambridge University Press.

Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. 2012. RockSalt: Better, Faster, Stronger SFI for the x86. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*. ACM. https://doi.org/10.1145/2254064.2254111

Greg Morrisett, David Walker, Karl Crary, and Neal Glew. 1999b. From System F to Typed Assembly Language. *ACM Transactions on Programming Languages and Systems* 21 (May 1999), 527–568. https://doi.org/10.1145/319301.319345

Mozilla. 2021. Firefox Public Data Report. https://data.firefox.com/dashboard/hardware.

Nacl Issue 1607 2011. Issue 1607: Signal handling change allows inner sandbox escape on x86-32 Linux in Chrome. https://bugs.chromium.org/p/nativeclient/issues/detail?id=1607.

Nacl Issue 1633 2011. Issue 1633: Inner sandbox escape on 64-bit Windows via KiUserExceptionDispatcher. https://bugs.chromium.org/p/nativeclient/issues/detail?id=1633.

Nacl Issue 2919 2012. Issue 2919: Security: NaClSwitch() leaks NaClThreadContext pointer to x86-32 untrusted code. https://bugs.chromium.org/p/nativeclient/issues/detail?id=2919.

Nacl Issue 775 2010. Issue 775: Uninitialized sendmsg syscall arguments in sel_ldr. https://bugs.chromium.org/p/nativeclient/issues/detail?id=775.

Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, Srdjan Capkun and Franziska Roesner (Eds.). USENIX Association, 699–716.

Shravan Narayan, Tal Garfinkel, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2019. Gobi: WebAssembly as a Practical Path to Library Sandboxing. arXiv:1912.02285 [cs.CR]

Native Client team. 2009. Native Client security contest archive. https://developer.chrome.com/docs/native-client/community/security-contest/.

Ben Niu and Gang Tan. 2014. RockJIT: Securing just-in-time compilation using modular control-flow integrity. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. ACM. https://doi.org/10.1145/2660267.2660281

Mathias Payer and Thomas R. Gross. 2011. Fine-Grained User-Space Security through Virtualization. In *Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. Association for Computing Machinery, New York, NY, USA. https://doi.org/10.1145/1952682.1952703

Weizhong Qiang, Yong Cao, Weiqi Dai, Deqing Zou, Hai Jin, and Benxi Liu. 2017. Libsec: A Hardware Virtualization-Based Isolation for Shared Library. In *2017 IEEE 19th International Conference on High Performance Computing and Communications; IEEE 15th International Conference on Smart City; IEEE 3rd International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. 34–41. https://doi.org/10.1109/HPCC-SmartCity-DSS.2017.5

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

David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. 2020. Donky: Domain Keys – Efficient In-Process Isolation for RISC-V and x86. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1677–1694. https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel

David Sehr, Robert Muth, Karl Schimpf, Cliff Biffle, Victor Khimenko, Bennet Yee, Brad Chen, and Egor Pasko. 2010. Adapting Software Fault Isolation to Contemporary CPU Architectures. In *19th USENIX Security Symposium, Washington, DC, USA, August 11-13, 2010, Proceedings*. USENIX Association, 1–12.

Margo I Seltzer, Yasuhiro Endo, Christopher Small, and Keith A Smith. 1996. Dealing with disaster: Surviving misbehaved kernel extensions. In *Proceedings of the Second USENIX Symposium on Operating Systems Design and Implementation (OSDI), Seattle, Washington, USA, October 28-31, 1996*. ACM. https://doi.org/10.1145/238721.238779

Joseph Siefers, Gang Tan, and Greg Morrisett. 2010. Robusta: Taming the native beast of the JVM. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*. ACM. https://doi.org/10.1145/1866307.1866331

Lau Skorstengaard, Dominique Devriese, and Lars Birkedal. 2019. StkTokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities. *Proceedings of the ACM on Programming Languages* 3, POPL (Jan. 2019), 1–28. https://doi.org/10.1145/3290332

Gang Tan et al. 2017. *Principles and implementation techniques of software-based fault isolation*. Now Publishers.

The LLVM Foundation. 2018. *Automatic variable initialization*. https://reviews.llvm.org/rL349442

The LLVM Foundation. 2021a. *Control Flow Integrity, Clang 12 documentation*. https://clang.llvm.org/docs/ControlFlowIntegrity.html

The LLVM Foundation. 2021b. *SafeStack, Clang 12 documentation*. https://clang.llvm.org/docs/SafeStack.html

Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. 2014. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Security Symposium*, Kevin Fu and Jaeyeon Jung (Eds.). 941–955.

Alex Tsariounov. 2021. *Shielding Linux Resources—Introduction*. https://documentation.suse.com/sle-rt/15-SP1/html/SLE-RT-all/cha-shielding-intro.html

Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. 2019. ERIM: Secure, Efficient In-process Isolation with Protection Keys (MPK). In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, 1221–1238.

K. Varda. 2018. WebAssembly on Cloudflare workers. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/.

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

R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera. 2015. CHERI: A Hybrid Capability-System Architecture for Scalable Software Compartmentalization. In *2015 IEEE Symposium on Security and Privacy*. 20–37. https://doi.org/10.1109/SP.2015.9

Conrad Watt, Andreas Rossberg, and Jean Pichon-Pharabod. 2019. Weakening WebAssembly. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 133 (Oct. 2019), 28 pages. https://doi.org/10.1145/3360559

WebAssembly Community Group. 2021. *Exception Handling*. https://github.com/WebAssembly/exception-handling

Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *PLDI*.

Bennet Yee, David Sehr, Gregory Dardyk, J. Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2009. Native Client: A Sandbox for Portable, Untrusted x86 Native Code. In *2009 30th IEEE Symposium on Security and Privacy*. 79–93. https://doi.org/10.1109/SP.2009.25 ISSN: 2375-1207.

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

Bin Zeng, Gang Tan, and Greg Morrisett. 2011. Combining Control-Flow Integrity and Static Analysis for Efficient and Validated Data Sandboxing. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (Chicago, Illinois, USA) *(CCS '11)*. Association for Computing Machinery, New York, NY, USA, 29–40. https://doi.org/10.1145/2046707.2046713

Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. 2011. ARMor: fully verified software fault isolation. In *Proceedings of the 11th International Conference on Embedded Software, EMSOFT 2011, part of the Seventh Embedded Systems Week, ESWeek 2011, Taipei, Taiwan, October 9-14, 2011*. ACM. https://doi.org/10.1145/2038642.2038687

Yajin Zhou, Xiaoguang Wang, Yue Chen, and Zhi Wang. 2014. Armlock: Hardware-based fault isolation for arm. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, Scottsdale, AZ, USA, November 3-7, 2014*. ACM. https://doi.org/10.1145/2660267.2660344