

Isolation Without Taxation: Near Zero Cost Transitions for SFI

Matthew Kolosick* Shravan Narayan* Conrad Watt†
Michael LeMay† Deepak Garg§ Ranjit Jhala* Deian Stefan*

*UC San Diego †University of Cambridge ‡Intel Labs §Max Planck Institute for Software Systems

Abstract—Almost all SFI systems use heavyweight transitions that incur significant performance overhead from saving and restoring registers when context switching between application and sandbox code. We identify a set of *zero-cost conditions* that characterize when sandboxed code is well-structured enough so that security can be guaranteed via lightweight *zero-cost* transitions. We show that using WebAssembly (Wasm) as an intermediate representation for low-level code naturally results in a SFI system with zero-cost transitions, and modify the Lucet Wasm compiler and its runtime to use zero-cost transitions. Our modifications speed up image and font rendering in Firefox by up to 29.7% and 10% respectively. We also describe a new purpose-built fast SFI system, *SegmentZero32*, that uses x86 segmentation and LLVM with mostly off-the-shelf passes to enforce our zero-cost conditions. While this enforcement incurs some runtime cost within the sandboxed code, we find that, on Firefox image and font rendering benchmarks, the time saved per transition allows *SegmentZero32* to outperform even an idealized hardware isolation system where memory isolation incurs zero performance overhead but the use of heavyweight transitions is required.

I. INTRODUCTION

Software-based fault isolation (SFI) is a lightweight alternative to process-based isolation, which isolates untrusted code with runtime checks that restrict (sandbox) the code to a specific region of the address space [1], [2]. Though SFI runtime checks typically slow down the code running in the sandbox, in application domains where sandboxed “components are tightly coupled and require frequent domain crossings,” the low overhead of SFI transitions more than makes up for the added cost [3]. For example, SFI has been used to isolate code in OS kernels [4]–[7], browsers [8]–[10], runtime systems [11]–[13], and storage systems [1], [14], [15].

More recently, Mozilla started using WebAssembly (Wasm) based SFI to sandbox third-party C libraries in Firefox [16], [17], and companies like Fastly are using Wasm to isolate tenant code on their edge clouds [18]. SFI allows Mozilla to isolate libraries like *libgraphite* (font shaping), *libexpat* (XML parsing), and *hunspell* (spell checking) that are tightly coupled and process content in a streaming fashion—

and thus require frequent domain crossings. Similarly, SFI allows Fastly to service thousands of tenants per second (per core), each of which calls into the runtime multiple times when handling a request—all within fractions of a millisecond [19].

While there have been significant strides on SFI enforcement (e.g., on x86 [2], [9], [14], [20], x86-64 [21], SPARC® [22], and ARM® [21], [23], [24]), context switching in SFI systems remains largely unexplored. Since Wahbe et al.’s original work [1], almost all SFI systems have used *heavyweight transitions* for context switching. These transitions switch domains by tying into the underlying SFI enforcement mechanism. For example, when transitioning into a sandbox they might set segment registers [9] or memory protection keys [25], [26] to ensure the sandbox code is memory isolated. They also save, scrub, and restore machine state (e.g., the stack pointer, program counter, and callee-save registers) to ensure confidentiality and integrity. This code is not only hard to get right (e.g., it must account for different architectures, platforms, and their quirks [27]) but also has significant overheads (§VI).

In this paper we revisit context switching in SFI systems and realize the 90’s vision of reducing the cost a context switch to (roughly) that of a function call. Through five contributions we show how to design SFI systems with near zero-cost transitions:

1. Formal model of secure transitions (§III). Simply removing heavyweight transitions for many SFI systems is unsafe; without transitions an attacker can easily escape the SFI sandbox. Thus, our first contribution is the first formal, declarative, and high-level model that elucidates the role of transitions in making SFI secure (§II). Intuitively, secure transitions protect the integrity and confidentiality of machine state across the domain transition and provide *well-bracketed* control flow, i.e., that returns actually return to their call sites.

2. Zero-cost conditions for SFI (§IV). Heavyweight transitions provide security by wrapping calls and returns to ensure that sandboxed code cannot, for example, read secret registers or tamper with the stack pointer. These transitions are necessary when the code running in the sandbox is arbitrary native code. In practice, though, most SFI systems enforce some structure on sandboxed code. For example, NaCl uses control-flow integrity (CFI) to restrict the sandbox’s control flow to its own code region, and requires dynamic checks on reads and writes [3], [9], [10]. This structure simplifies SFI enforcement and verification (that the enforcement is correct). Our insight is that imposing structure on sandboxed code also allows us

This work was supported in part by gifts from Cisco; by the NSF under Grant Number CNS-1514435, 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.

to replace heavyweight transitions with *zero-cost transitions* which are close to bare function calls.

This insight is inspired by work on *language-based isolation* [28]–[33]. Language-based systems like Singularity [32] use high-level, type- and memory-safe languages (e.g., Sing#) to isolate code at the language level. The compositional structure of such languages allow these systems to use simple function calls as secure cross-domain transitions: the *language* enforces well-bracketed control flow and local state encapsulation, i.e., confidentiality and integrity of machine state. We cannot realistically rewrite huge systems like Firefox (or even the third-party libraries in Firefox) in a high-level language [3], so we instead capture the essence of what makes it possible for language-based systems to safely use zero cost transitions and adapt this to the SFI setting.

Our second contribution precisely defines the *zero-cost conditions* that sandbox code must satisfy to safely use zero-cost transitions: Sandboxed code must follow a type-directed CFI discipline, have well-bracketed control flow, enforce local state (stack and register) encapsulation, and ensure registers and stack slots are initialized before use. These conditions are only slightly more onerous than the structure some SFI systems already require of sandboxed code (e.g., compared to NaCl, we rely on type-based CFI instead of coarse-grained CFI and use a safe stack instead of a separate stack). We state the zero-cost conditions in terms of a dynamic overlay semantics which we can instantiate to model different SFI systems.

3. Instantiating the zero-cost model (§V). Our third contribution is an instantiation of our zero-cost model to two SFI systems: Wasm and SegmentZero32.

Wasm is a low-level bytecode with a sound, static type system designed to be targeted by compilers for languages such as C [10]. We study Wasm as an intermediate representation (IR) for SFI [16], [34]–[36], i.e., compiling untrusted C/C++ libraries to native binaries using Wasm as an IR, that are then linked against an application like Firefox (which is not compiled through Wasm). We show that Wasm satisfies our zero-cost conditions—and replace the heavyweight transitions used by existing Wasm SFI compilers with zero-cost transitions.

We design the SegmentZero32 SFI system to (1) use 32-bit x86 segmentation hardware to enforce (heap) memory isolation and (2) restrict the structure of code according to our zero-cost conditions. Unlike previous SFI systems that use segmentation [2], [9], [14], we use a set of mostly off-the-shelf LLVM compilation passes and one custom compilation pass (Section V-B) to enforce our zero-cost conditions, including type-based CFI, and ensure that SegmentZero32 can safely use zero-cost transitions. While the prevalence of 32-bit x86 systems is declining, it nevertheless still constitutes over 20% of the Firefox web browser’s user base (around 45 million users)¹; SegmentZero32 would allow for high performance library sandboxing on these machines.

4. Proofs of security. Our fourth contribution is a set of proofs of security. We prove that NaCl’s heavyweight transitions

are secure (§B) and that, when the zero-cost conditions are met, zero-cost transitions are secure (§C). We also prove that Wasm meets the zero-cost conditions and can therefore safely elide heavyweight transitions while maintaining integrity and confidentiality (§D).

5. Implementation and evaluation (§VI). Our last contribution is an implementation and evaluation of Wasm and SegmentZero32 with zero-cost transitions. We integrate the two SFI systems into the RLBox sandboxing framework [16], and evaluate the performance of zero-cost transitions on several microbenchmarks and two macrobenchmarks—image decoding and font rendering in Firefox, which are currently sandboxed using Wasm-based SFI. We find that zero-cost transitions speed up Wasm-sandboxed image decoding by (up to) 29.7% and font rendering by 10%. SegmentZero32 imposes an overhead of (at most) 24% on image decoding and 22.5% on font rendering relative to unsandboxed native code. These overheads are significantly lower than even an idealized hardware isolation system where memory isolation adds zero overhead but the use of heavyweight transitions is required.

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

II. 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.

A. The need for secure transitions

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 }
```

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 (e.g., as in [25]), 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

¹See <https://data.firefox.com/dashboard/hardware>, last visited March 2021

the register `r12`. In the System V calling convention `r12` is a *callee-saved* register [37], 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, 0
2 ret
```

Since `r12` (screen) is the used by Firefox 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 and, thus, 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 (e.g., as in [25]), the attacker can 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.

B. Heavyweight transitions

SFI toolchains — from NaCl [9] to Wasm native compilers like Lucet [38] and WAMR [39] — use *heavyweight transitions* to wrap calls and returns and prevent the aforementioned attacks. 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 the 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 the sandboxed code cannot access arguments stored on the application stack. Finally, the trampoline tracks the actual return address to return 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 (§VI). 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. Implementation mistakes can — and have [40]–[45] — resulted in sandbox escape attacks.

C. 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 of known types. 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. Type-directed forward-edge CFI. First, our monitor requires that the library code enforces type-directed forward-edge CFI. That is, for every call instruction encountered during execution, the jump target address is the start of one of the library functions and the number and types of arguments expected by that function match what are actually passed. This provides two main properties that are critical for security. First, it ensures that each function starts from a (statically) known stack shape, preventing a class of attack 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. Second, it provides the structure needed to define a property capturing proper restoring of callee-save registers (discussed in point three below).

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. Unrestricted control flow is at odds with compositional reasoning. It also makes it difficult to define correct restoration of callee-save registers since it is unclear what “returning” from a function call means. 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. Callee-save registers restoration. Building on well-bracketed control flow and, in particular, the resulting definition of the beginning and end of a function call, we define function-level adherence to callee-save register conventions: our monitor tracks callee-save state and checks that it has been correctly restored at every return. 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.

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.

We prove that these five zero-cost conditions characterize libraries that can be securely isolated with zero-cost transitions, using the combination of memory isolation and function calls without the overhead of springboards and trampolines (see Theorem 1). In Section V-A we show that the Wasm type

	n	\in	\mathbb{N}
$Priv$	\ni	p	$::= \text{app} \mid \text{lib}$
Val	\ni	v	$::= \langle n, p \rangle$
Reg	\ni	r	$::= r_n \mid sp \mid pc$
$Region$	\ni	k	$\in \mathbb{N} \rightarrow \mathbb{N}$
$Immediate$	\ni	i	$::= r \mid v \mid i \oplus i$
$Command$	\ni	c	$::= r \leftarrow \text{pop}_p \mid \text{push}_p i \mid \text{jmp}_k i \mid$ $r \leftarrow \text{load}_k i \mid \text{store}_k i := i \mid$ $\text{gatecall}_n i \mid \text{gateret}$ $r \leftarrow \text{mov } i \mid \text{call}_k i \mid \text{ret}_k$ $\text{gatecall}_n i \mid \text{storelabel}_p i$
$Code$	\ni	C	$::= \mathbb{N} \rightarrow Priv \times Command$
$RegVals$	\ni	R	$::= Reg \rightarrow Val$
$Memory$	\ni	M	$::= \mathbb{N} \rightarrow Val$
$State$	\ni	Ψ	$::= \text{error} \mid$ $\{pc : \mathbb{N}, sp : \mathbb{N}, R : RegVals,$ $M : Memory, C : Code\}$

Figure 1: Syntax

system is strict enough to ensure that a Wasm compiler generates native code that meets these conditions. Finally, in Section V-B 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.

III. 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 memory (M) are separated, and, to capture the separation 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 $\Psi \langle c \rangle_p$ for $\Psi.C(\Psi.pc) = (p, c)$, that is that the current instruction is c in security domain p . We write $\Psi_0 \in Program$ 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 `gatecalln i` and `gateret`, that serve as the *only* way to switch

between the two security domains. The first, `gatecalln i`, 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 $f(n)|_{n \in M_{lib}} = n$.

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 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 ($\Psi \rightarrow \Psi'$). The rules are standard; we show the rule for load here:

$$\frac{\langle addr \rangle = \mathcal{V}_\Psi(i) \quad addr' = k(addr) \quad v = \Psi.M(addr') \quad R' = \Psi.R[r \mapsto v]}{\Psi(r \leftarrow \text{load}_k i) \rightarrow \Psi^{++}[R := R']}$$

$\mathcal{V}_\Psi(i)$ evaluates the immediate value based on the register file and Ψ^{++} increments pc , checking that it remains within the same security domain. 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 .

A. Secure transitions

Our goal is to specify when sandboxed code has enough structure to elide springboards and trampolines while maintaining the same security guarantees. One way to do so, is by stipulating that zero-cost sandboxed code is *equivalent* with or without springboards and trampolines. Unfortunately, such a specification would be unpleasantly *operational*. First, it would be hard to get right: springboards and trampolines are tricky and involve a significant amount of low-level implementation detail. Second, such a specification would be platform-specific as each architecture and calling-conventions requires different springboards and trampolines.

Instead, 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 for studying

$$\frac{\Psi_1 \rightarrow \Psi_2 \quad \Psi_1(c_1)_{p_1} \quad \Psi_2(c_2)_{p_2} \quad p_1 = p_2 = p}{\Psi_1 \xrightarrow{p} \Psi_2} \quad \frac{\Psi \xrightarrow{p} \Psi'}{\Psi \xrightarrow{\square} \Psi'} \quad \frac{\Psi \xrightarrow{wb} \Psi'}{\Psi \xrightarrow{\square} \Psi'}$$

$$\frac{\Psi \rightarrow \Psi_1 \xrightarrow{\square}^* \Psi_2 \rightarrow \Psi' \quad \Psi(\text{gatecall}_n i) \quad \Psi_2(\text{gateret})}{\Psi \xrightarrow{wb} \Psi'}$$

Figure 2: Well-Bracketed Transitions

systems that enable zero-cost transitions and for exploring the correctness of particular implementations of springboards and trampolines. As a baseline we prove that NaCl-style heavyweight transitions satisfy the high-level properties.

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 entering and exiting the sandbox. 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 $\Psi \xrightarrow{wb} \Psi'$ (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 balanced 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 internal 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 an integrity property 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

$\mathcal{I} : \text{Trace} \times \text{State} \times \text{State} \rightarrow \mathbb{P}$ 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 $\mathcal{I} : \text{Trace} \times \text{State} \times \text{State} \rightarrow \mathbb{P}$. Then, if $\Psi_0 \in \text{Program}$, $\pi = \Psi_0 \rightarrow^* \Psi_1$, $\Psi_1(_)_{\text{app}}$, and $\Psi_1 \xrightarrow{wb} \Psi_2$ imply that $\mathcal{I}(\pi, \Psi_1, \Psi_2)$, we say that an SFI transition system has \mathcal{I} -integrity.*

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 ignores the *Trace* argument but requires the callee-save registers' values to be equal in both states:

Definition 2 (Callee-Save Register Integrity). *Let CSR be the callee-save registers and define $\text{CSR}(_, \Psi_1, \Psi_2) \triangleq \Psi_2.R(\text{CSR}) = \Psi_1.R(\text{CSR})$. If an SFI transition system has 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 can similarly be 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 data on the stack. 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 $\text{return-address}(\pi)$ (details in the appendix, see Figure 16) that computes the set of locations of return addresses based on 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 make sure that secrets cannot be leaked to the untrusted library, that is they must provide *confidentiality*. We specify confidentiality as non-interference: “changing secret inputs does not affect public outputs.” In the context of library sandboxing we associate “secret” with application data and “non-secret” with library data, as the purpose of library sandboxing is isolating untrusted components.² We thus augment values with labels *app* or *lib* where $\text{lib} \sqsubseteq \text{app}$ (non-secret *can* flow to secret) and $\text{app} \not\sqsubseteq \text{lib}$ (secret *cannot* “flow to” non-secret). Values are

then a pair of a \mathbb{N} and a label p . We also extend our instruction set with $r \leftarrow \text{movlabel}_p$ and $\text{storelabel}_p i$ that allow the application to dynamically assign the label p to the value stored in register r or pointed to by i , respectively.

Next we ask what comprises a public output: that is, what should be considered as *leaking* the secret (app-labeled) data? In these low-level systems, leaks occur when secret data could be written to a file or exfiltrated over the network. However, sandboxed libraries aren't given direct access to system calls that would enable such exfiltration and are only given indirect access through callbacks provided by the host application. We thus over-approximate the public outputs as the set of values *returned* to the application: this includes all the values that could be leaked. The returned values include all arguments to a `gatecall` callback, the return value when doing a `gateret` to the application, and all values stored in the sandboxed heap (H_{lib}) which may be referenced by other returned values.

A further consideration that we must account for in defining a noninterference property is that, during a callback to the application, the application may choose to declassify additional information. For instance, a sandboxed image decoding library might, after parsing the file header, make a callback requesting the appropriate data to decode the rest of the image. This application callback will then transfer that data (which was previously confidential application data) over to the sandboxed memory, declassifying it in the transfer.

Due to the possibility of intentional declassification, we choose to follow [46] and define confidentiality as disjoint non-interference as follows. We use $\Psi =_{\text{lib}} \Psi'$ to mean that Ψ and Ψ' agree on all values with label *lib*, representing varying secret inputs. We further use $\Psi =_{\text{call } m} \Psi'$ for when Ψ and Ψ' agree on all sandboxed heap values, the program counter, and the m arguments passed to a callback and $\Psi =_{\text{ret}} \Psi'$ for when Ψ and Ψ' agree on all sandboxed heap values, the program counter, and the value in the return register (written r_{ret}).³ This lets us define noninterference as follows:

Definition 4 (Disjoint Noninterference).

If, for all $\Psi_0 \in \text{Program}$, $\Psi_1(_)_{\text{lib}}$, $\Psi_3(_)_{\text{app}}$, $\Psi_0 \rightarrow^ \Psi_1 \xrightarrow{\text{lib}}^n \Psi_2 \rightarrow \Psi_3$, and, for all Ψ'_1 such that $\Psi_1 =_{\text{lib}} \Psi'_1$, we have that $\Psi'_1 \xrightarrow{\text{lib}}^n \Psi'_2 \rightarrow \Psi'_3$, $\Psi'_3(_)_{\text{app}}$, $\Psi_3.pc = \Psi'_3.pc$, and 1) $\Psi_2(\text{gatecall}_m i)$, $\Psi'_2(\text{gatecall}_m i)$, and $\Psi_3 =_{\text{call } m} \Psi'_3$ or 2) $\Psi_2(\text{gateret})$, $\Psi'_2(\text{gateret})$, and $\Psi_3 =_{\text{ret}} \Psi'_3$, then we say that the SFI transition system has the disjoint noninterference property.*

This definition states that, for any consecutive sequence of executing exactly n steps within the sandbox 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 transition system satisfying Disjoint Noninterference is guaranteed to not declassify new data while running within the sandbox.

²This could also be extended to a setting with mutually distrusting components.

$$\begin{aligned}
\text{Frame} \ni SF &::= \{ \text{base} : \mathbb{N} \\
&\quad \text{ret-addr-loc} : \mathbb{N} \\
&\quad \text{csr-vals} : \wp(\text{Reg} \times \mathbb{N}) \} \\
\text{Function} \ni F &::= \{ \text{instrs} : \mathbb{N} \rightarrow \text{Command} \\
&\quad \text{entry} : \mathbb{N} \\
&\quad \text{type} : \mathbb{N} \} \\
\text{oState} \ni \Phi &::= \text{oerror} \\
&\quad | \{ \Psi : \text{State} \\
&\quad \quad \text{funcs} : \mathbb{N} \rightarrow \text{Function} \\
&\quad \quad \text{stack} : [\text{Frame}] \}
\end{aligned}$$

Figure 3: oSFIasm Extended Syntax

IV. ZERO-COST TRANSITION CONDITIONS

We define our zero-cost conditions as a safety monitor with a language oSFIasm overlaid on top of SFIasm. This language 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 that our integrity and confidentiality properties are maintained.

Syntax of oSFIasm. Figure 3 shows the extended syntax of oSFIasm. 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 ($\Phi.\text{funcs}$). $\Phi.\text{funcs}$ maps each command in the sandboxed library to its parent function. Functions (F) also store the code indices of their commands as the field $F.\text{instrs}$, store the entry point ($F.\text{entry}$), and track the number of arguments the function expects ($F.\text{type}$). This partitioning of sandboxed code into functions is static. Second, the overlay state dynamically tracks a list of overlay stack frames ($\Phi.\text{stack}$). 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 ($SF.\text{base}$), the stack location of the return address ($SF.\text{ret-addr}$), and the values of the callee save registers upon entry to the function ($SF.\text{csr-vals}$). 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**.

A. 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

$$\begin{aligned}
\text{OCALL} & \frac{\langle n, \text{lib} \rangle = \mathcal{V}_\Phi(i) \quad n' = k(n) \quad sp' = \Phi.sp + 1 \\
M' = \Phi.M[sp' \mapsto \Phi.pc + 1] \quad \text{stack}' = [SF] \uparrow \Phi.stack \\
SF = \text{new-frame}(\Phi, n', sp') \quad \text{typechecks}(\Phi, n', sp') \\
\Phi' = \Phi[\text{stack} := \text{stack}', pc := n', sp := sp', M := M']}{\Phi(\text{call}_k i)_{\text{lib}} \rightsquigarrow \Phi'} \\
\text{ORET} & \frac{\text{is-ret-addr}(\Phi, \Phi.sp) \quad \langle n \rangle = \Phi.M(\Phi.sp) \quad n' = k(n) \\
\text{csr-restored}(\Phi) \quad \Phi' = \text{pop-frame}(\Phi)}{\Phi(\text{ret}_k)_{\text{lib}} \rightsquigarrow \Phi'[pc := n', sp := \Phi.sp - 1]} \\
\text{OJMP} & \frac{\langle n, \text{lib} \rangle = \mathcal{V}_\Phi(i) \\
n' = k(n) \quad \text{in-same-func}(\Phi, \Phi.pc, n')}{\Phi(\text{jmp}_k i)_{\text{lib}} \rightsquigarrow \Phi[pc := n']} \\
\text{OSTORE} & \frac{\langle n, \text{lib} \rangle = \mathcal{V}_\Phi(i) \quad v = \langle _, p_{i'} \rangle = \mathcal{V}_\Phi(i') \\
M' = \Phi.M[n' \mapsto v] \quad \text{writeable}(\Phi, n') \\
n' = k(n) \quad p_{i'} = \text{app} \implies n' \notin H_{\text{lib}}}{\Phi(\text{store}_k i := i')_{\text{lib}} \rightsquigarrow \Phi^{++}[M := M']}
\end{aligned}$$

Figure 4: oSFIasm Operational Semantics Excerpt

$$\begin{aligned}
& \frac{F = \Phi.\text{funcs}(\text{target}) \quad F.\text{entry} = \text{target} \quad sp \in S_p \\
[SF] \uparrow _ = \Phi.stack \quad sp \geq SF.\text{ret-addr} + F.\text{type}}{\text{typechecks}(\Phi, \text{target}, sp)} \\
& \frac{[SF] \uparrow _ = \Phi.stack \quad F \in \text{cod}(\Phi.\text{funcs}) \\
\text{ret-addr} = SF.\text{ret-addr} \quad n, n' \in F.\text{instrs}}{\text{is-ret-addr}(\Phi, \text{ret-addr}) \quad \text{in-same-func}(\Phi, n, n')} \\
& \frac{[SF] \uparrow _ = \Phi.stack \\
\forall (r, n) \in SF.\text{csr-vals}. \Phi.R(r) = n}{\text{csr-restored}(\Phi)} \\
& \frac{[SF] \uparrow _ = \Phi.stack \\
n \in S_p \implies n \geq SF.\text{base} \wedge n \neq SF.\text{ret-addr}}{\text{writeable}(\Phi, n)}
\end{aligned}$$

Figure 5: oSFIasm Semantics Auxiliary Predicates

(with auxiliary definitions shown in Figure 5) shows an excerpt of the checks, which we describe below. Full definitions can be found in Appendix C.

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) 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

³Full definitions are in Appendix A Definition 8 and Definition 9.

stack. On top of this check, `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 there is 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. `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 $p_{i'} = \text{app} \Rightarrow n' \notin H_{\text{lib}}$ 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.

B. 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 theorems showing that the overlay semantics is sound with respect to each of our security properties; we show the statement for callee-save register integrity below.

Theorem 1 (Overlay Callee-Save Register Soundness). *If $\Phi_0 \in \text{Program}$, $\Phi_0 \rightsquigarrow^* \Phi_1$, $\Phi_1.p = \text{app}$, and $\Phi_1 \rightsquigarrow^* \Phi_2$ such that $\Phi_2 \neq \text{oerror}$ and $\Phi_1.\Psi \xrightarrow{wb} \Phi_2.\Psi$, then $\Phi_2.R(\text{CSR}) = \Phi_1.R(\text{CSR})$.*

V. INSTANTIATING ZERO-COST

We describe two isolation systems that securely support zero-cost transitions: they meet the overlay monitor zero-cost conditions. The first is an SFI system using `WebAssembly` as an IR before compiling to native code using the `Lucet` toolchain [38]. Here we rely on the language-level invariants of `Wasm` to satisfy our zero-cost requirements. The second, `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.

A. 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, in combination with a trusted compiler, produce binaries satisfying the conditions required to support secure zero-cost transitions. 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 the proper 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 [47]. 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 heavy-weight 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.

Proving Wasm secure. We prove that compiled Wasm libraries can safely elide springboards and trampolines while maintaining integrity and confidentiality, by showing that the compiled code would not violate the safety monitor. This allows us to apply Theorem 1 and its analogues. It is relatively straightforward (with one exception) to verify that the properties of Wasm as described above satisfy the necessary safety conditions, by showing that the linear instructions of each basic block within a function are safe and that each basic block properly initializes the local state for any block it jumps to. Type-safe execution of calls is guaranteed by Wasm’s required type checking, and confidentiality is preserved by the language invariant that all registers and stack slots are either written before use or zero-initialized.

The crucial exception in the proof is 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 Appendix D-A) 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 locally prove 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 2 (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 and Theorem 1:

Theorem 3 (Adequacy of Wasm Logical Relation). *For any number of steps $n \in \mathbb{N}$, library L such that $(n, L) \in \mathcal{L}$, program $\Phi_0 \in \text{Program}$ using L , and $n' \leq n$, if $\Phi_0 \rightsquigarrow^{n'} \Phi'$ then $\Phi' \neq \text{oerror}$.*

Details of the logical relation and proofs are in Appendix D.

B. SegmentZero32

SegmentZero32 is our novel design of a 32-bit SFI scheme developed using the zero-cost conditions as a design guide. SegmentZero32 leverages Clang/LLVM compiler passes to directly enforce the structure required for zero-cost transitions on C code, rather than relying on Wasm as an IR. Similar to NaCl [9] and vx32 [14], SegmentZero32 takes advantage of the x86 segmented memory model [48] for memory isolation. Segmentation allows programs to demarcate the memory regions that are used as the stack, data (heap), and code regions, with hardware support for range checks. However, both NaCl and vx32 employ only a single memory range that is shared by the sandbox stack and heap. SegmentZero32 instead separates these two, and then leverages existing exploit mitigation passes (along with a pass of our design) to provide isolation while supporting zero-cost transitions. We describe these passes and how we use them below:

Protecting the stack. Simply using separate stack and non-stack segments does not protect the return addresses and context saved on the stack from a standard buffer overflow attack. We solve this issue by applying the SafeStack [49], [50] compiler pass, an LLVM pass designed to protect against stack corruption. The pass splits 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.⁴ All other stack values else are moved to the heap.

While the above modifications are sufficient for stack safety, in practice we ran into a compatibility challenge when using the stack segment to access the safe stack: code emitted by LLVM may use the data segment when accessing the safe stack because the compiler assumes a flat, non-segmented memory model. To correct this, we develop an additional LLVM pass that annotates instructions with stack and data segment override prefixes as

⁴We also employ LLVM’s existing stack-heap clash detection (flag `-fstack-clash-protection`) to prevent the stack growing into the heap.

needed. The pass assumes that only `esp` points to the safe stack at the start of each function. We then track the flow of addresses derived from `esp` to other registers throughout the function to determine whether any given memory operand refers to the safe stack. If a memory operand references the incorrect segment, the pass emits a segment override prefix.

With this issue patched, all heap and global loads and stores are statically assigned a non-stack segment register (in our model this appears as a guard $f(n)$ with codomain H_{lib}) and all stack operations use verifiably safe offsets and thus can remain unguarded. This guarantees both local state isolation and protects saved register values, the stack pointer, and the return address. Notably, we achieve these properties *deterministically* as the segmentation based memory isolation ensures that, even if an attacker guesses the stack location, they cannot dereference a pointer to the stack.

Control flow. The above protects the callee-save registers and the return address saved in each sandbox call frame, guaranteeing the restoration of callee-save registers and well-bracketing *iff forward control flow is enforced*. Fortunately, enforcing forward edge CFI has been widely studied [51]. We use a CFI pass as implemented in Clang/LLVM [52], [53] including flags to dynamically protect indirect function calls, ensuring forward control flow integrity. Further, `SegmentZero32` conservatively bans non-structured control flow (including `setjmp/longjmp`) in the C source code. A more permissive approach is possible in principle, akin to the `FixIrreducibleControlFlow` LLVM pass already used in WebAssembly compilation, but we leave this for future work.

Confidentiality. To guarantee confidentiality we employ an existing (experimental) Clang pass that ensures all local variables (i.e., stack slots or registers) are auto initialized [54]. This ensures that scratch registers cannot leak secrets as all sandbox values are written before use. This demonstrates how our zero-cost transition scheme enables a more platform agnostic defense: we do not need to account for the details of what scratch registers a processor has, as *no* application values can flow to sandbox variables.

We do not include an explicit proof of security for `SegmentZero32` as it closely follows the proof of Wasm security. The main differences are administrative changes to the logical relation definitions and a different control-flow graph for the blocks within each compiled function.

VI. EVALUATION

We evaluate zero-cost transitions by asking three questions:

- ▶ What is the performance overhead of zero-cost and heavyweight transitions in different SFI systems? (§VI-A)
- ▶ Do zero-cost transitions improve the end-to-end performance of applications that use WebAssembly SFI? (§VI-B)
- ▶ Can our `SegmentZero32` isolation scheme outperform isolation schemes which require heavyweight transitions (e.g., Native Client) in real workloads? (§VI-C)

Implementation. To answer these questions we study the performance of native (unsandboxed) code and seven isolation scheme implementations.

The first four isolation builds vary the transition models for a WebAssembly SFI system based on the Lucet [38] compiler; this lets us understand the benefits of zero-cost transitions in WebAssembly-based SFI systems. We investigate the following builds: the `WasmLucet` build uses the original heavyweight springboards and trampolines shipped with the Lucet runtime written in Rust. `WasmHeavy` adopts techniques from NaCl’s implementations and uses optimized assembly instructions to save and restore application context during transitions. `WasmZero` implements our zero-cost transition system, meaning transitions are simple function calls that do not perform any additional register saving/restoring or stack switching (library and application code execute on the same stack). In order to distinguish between the overhead of register saving/restoring and stack switching, we also test a `WasmReg` build which saves/restores registers similar to `WasmHeavy`, but shares the library and application stack like `WasmZero`.

The next three builds use different hardware based isolation schemes to measure whether a purpose-built zero-cost isolation scheme (`SegmentZero32`) can outperform state-of-the-art isolation schemes that do not support zero-cost transitions. Since `SegmentZero32` uses segmentation—a hardware feature supported only in 32-bit x86 programs—the next three builds are all run in 32-bit mode for a fair comparison. `SegmentZero32`, is the zero-cost segmentation scheme described in §V-B. `NaCl32`, is the Native Client 32-bit isolation scheme [9], as modified by Narayan et al. [16] to support library isolation; this scheme also leverages segmentation but employs transitions similar to `WasmHeavy`. We also compare against `IdealHeavy32`, an “optimal” hardware isolation scheme that is not zero-cost compatible; this scheme incurs no slowdowns to enforce, enable, or disable isolation. To simulate the performance of `IdealHeavy32`, we simply measure the performance of native code with heavy-weight trampolines.

We integrate each SFI scheme into Firefox using the `RLBox` framework [16]. `RLBox` already provides plugins for the `WasmLucet` and `NaCl32` builds, we implement the plugins for the remaining builds above.⁵

Machine / software setup. We run all benchmarks on a machine with Intel® Core™ i7-6700K with four 4GHz cores, 64GB RAM, running Ubuntu 20.04.1 LTS (kernel version 5.4.0-58). Benchmarks are run with a shielded isolated `cpuset` [55] consisting of one core with hyperthreading disabled and the clock frequency pinned to 2.2GHz. Wasm sandboxed code is generated with a two-part toolchain—C/C++ is first compiled to the Wasm format with Clang-11, and then to native code using the fork of the Lucet used by `RLBox` (snapshot from Dec 9th 2020). NaCl sandboxed code is generated with a modified

⁵While the `SegmentZero32` plugin accurately model transition costs, we emphasize that this is not production ready. In particular, we relax the restrictions on accessible address ranges as we need to allow access to static data sections in ELF binaries.

Wasm build	Direct call	Indirect call	Callback	Syscall
Func (in C)	1ns	56ns	56ns	24ns
WasmLucet	N/A	1137ns	N/A	N/A
WasmHeavy	120ns	209ns	172ns	192ns
WasmReg	120ns	210ns	172ns	192ns
WasmZero	7ns	66ns	67ns	60ns

32-bit hardware isolation build	Direct call	Indirect call	Callback	Syscall
Func (in C, 32-bit)	1ns	74ns	74ns	37ns
IdealHeavy32	324ns	179ns	154ns	181ns
NaCl32	N/A	714ns	373ns	356 ns
SegmentZero32	24ns	108ns	80ns	88ns

Figure 6: Costs of transitions in different isolation models. Zero-cost transitions are shown in **boldface**. Func is the performance of vanilla unsandboxed C to serve as a baseline.

version of Clang-4. All other C/C++ source code including SegmentZero32 sandboxed code as well as the benchmarks applications are compiled with Clang-11. We implement our Firefox benchmarks on a top of Firefox Nightly (from August 22, 2020).

A. Transition microbenchmarks

We measure the cost of different uses of transitions—direct and indirect calls into the sandbox, callbacks from the sandbox and syscall invocations from the sandbox—for the different isolation builds described. 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 6. All numbers are averages of one million repetitions, and repeated runs have negligible standard deviation. Note that Lucet’s and NaCl’s existing implementations do not support direct sandbox calls and Lucet also does not support custom callbacks or invocation of syscalls, so we do not report these numbers.

Among Wasm-based SFI schemes, zero-cost transitions (WasmZero) are significantly faster than even optimized assembly instructions implementing heavy-weight transitions (WasmHeavy). Lucet’s existing indirect calls written in Rust (WasmLucet) are significantly slower than both. Stack switching (the difference of WasmHeavy and WasmReg) adds a very small overhead for transitions. The performance of Func and WasmZero should be identical but is not. This is *not* because our transitions have a hidden cost; rather, it is because we are comparing code produced by two different compilers: Func is native code produced by Clang, while WasmZero is code produced by Lucet, and Lucet’s code generation is not yet highly optimized [56]. For example, in the benchmark that adds two numbers, Clang eliminates the function prologue and epilogue that save and restores the frame pointer, while Lucet does not. For hardware-based isolation, we compare our zero-cost transitions in SegmentZero32 to heavyweight transitions in NaCl32 and IdealHeavy32. We find that SegmentZero32 transitions are much faster

than IdealHeavy32 and NaCl32 transitions and only 23ns slower than Func for direct calls. SegmentZero32 incurs some overhead over WasmZero as hardware isolation schemes like SegmentZero32 and NaCl32 must invoke extra instructions to enable or disable the hardware based memory isolation in their transitions.

B. Zero-cost transitions for WebAssembly SFI

We evaluate the end-to-end performance impact of the different transition models using two libraries currently sandboxed in Firefox: font rendering with `libgraphite`, and image rendering with `libjpeg`. The performance of these sandboxed libraries was previously observed by Narayan et al. [16] to be particularly affected due to the high-number of transitions.

Font rendering. We evaluate the performance of `libgraphite` isolated with Wasm-based schemes using Kew’s benchmark⁶, which was also used by Narayan et al. The benchmark reflows the text on a page ten times, adjusting the size of the fonts each time to negate the effects of font caches. We run this benchmark 100 times and report the median execution time below (all values have standard deviations within 1%).

	WasmLucet	WasmHeavy	WasmReg	WasmZero
Font render	8173ms	2246ms	2230ms	2032ms

As expected, zero-cost transitions (WasmZero) result in the best performance. Compared to WasmZero, Lucet’s existing transitions slow down rendering by over 4×. Even when using the optimized transitions of WasmHeavy, performance is over 10% slower than WasmZero. Stack switching accounts for about 0.8% of WasmHeavy’s extra cost.

Image rendering. Firefox renders images in streaming mode, calling `libjpeg` to decode *every row of an image*. Hence, the amount of work done in our Wasm-sandboxed `libjpeg`, per call, is proportional to the width of the image, and the complexity of the rendering a row of the image. We expect the effect of per-transition overhead to decrease with increasing image width and increasing image complexity. Our benchmark thus varies the width and image complexity.

Figure 7 shows the costs of rendering three kinds of jpeg images—a simple image consisting of a single color (SimpleImage), a stock image from the Image Compression benchmark suite⁷ (StockImage), and an image of random pixels (RandomImage). Each image is rendered 500 times. We report the median decoding time (standard deviations are all under 1%). The costs are normalized to WasmZero to highlight the relative overheads of different transitions as compared to our zero-cost transitions. The results of WasmLucet are included in the Appendix (Figure 26) because the rendering times are up to 9.2× longer than the other builds and skews our graphs. We instead focus on evaluating the overheads of optimized heavy transitions. The WasmLucet performance numbers do highlight the fact that care is needed in implementing optimized

⁶Available at https://jfkthame.github.io/test/udhr_urd.html

⁷Online: https://imagecompression.info/test_images/. Visited Dec 9, 2020.

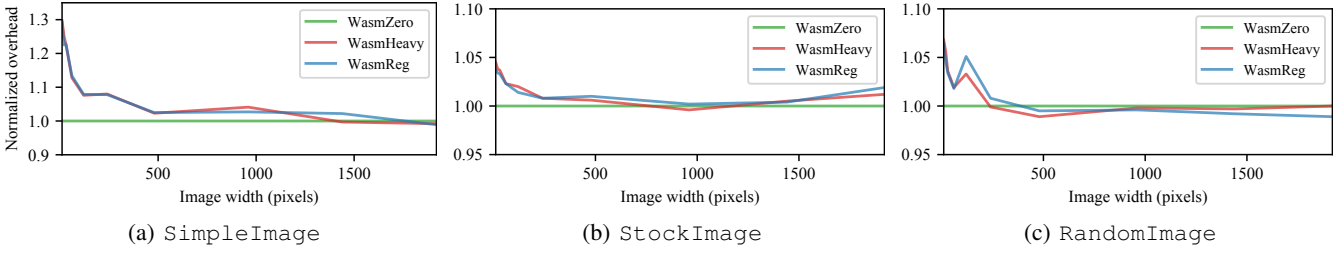


Figure 7: Performance of different wasm transitions on rendering of (a) a simple image with one color, (b) a stock image and (c) a complex image with random pixels, normalized to WasmZero. WasmZero transitions outperform other transitions. The difference diminishes with width, but narrower images are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

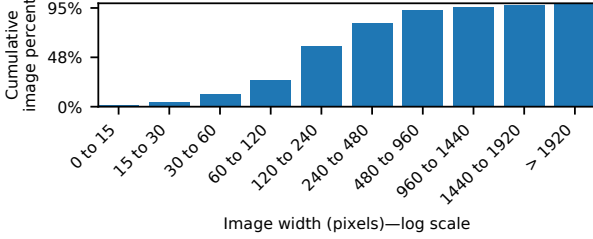


Figure 8: Cumulative distribution of image widths on the landing pages of the Alexa top 500 websites. Over 80% of the images have widths under 480 pixels. Narrower images have a higher transition rate, and thus higher relative overheads when using expensive transitions.

heavyweight transitions, and that the transitions available in current tools may not be fully optimized.

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 \times longer to render the image as with WasmZero transitions. However, this performance difference diminishes as image width increases. For StockImage and RandomImage, the WasmHeavy trends are similar, but the rendering time differences start at about 4.5%. However, 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, we find that most 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.

C. Zero-cost transitions for native isolation

In this section, we compare the performance of native code against code isolated with SegmentZero32 with zero-cost transitions, against code isolated with NaCl (NaCl32) which does not support zero-cost transitions. SegmentZero32 and NaCl32 isolate native-compiled libraries (without going through Wasm) and use hardware support to enforce isolation. Additionally, we also compare against a hypothetical isolation

with no isolation enforcement overhead (IdealHeavy32) with heavyweight transitions; this simulates the performance of isolation schemes using hardware such as MPK [25] which have negligible overhead but require heavyweight transitions. The IdealHeavy32 performance is simulated by running native unsandboxed code with heavyweight transitions.

To measure performance, we rerun our libgraphite and libjpeg benchmarks from the previous section, but isolate the libraries using SegmentZero32, NaCl32 and IdealHeavy32. 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 find that for these benchmarks, SegmentZero32 with zero-cost transitions outperforms NaCl32 as well as the hypothetical IdealHeavy32 isolation. We describe the individual benchmarks next.

Font rendering. The impact of these isolation schemes on font rendering is shown below (standard deviations under 1%).

	Unsandboxed 32-bit code	IdealHeavy32	NaCl32	SegmentZero32
Font render	1441ms	2399ms	2769ms	1765ms

We observe that NaCl32 and IdealHeavy32 incur overhead of 92% and 66% respectively compared to unsandboxed code. In contrast, SegmentZero32 only has an overhead of 22.5% as it does not have to save and restore registers or switch stacks. The overhead of SegmentZero32 over native code itself is due to a few different factors: first, SegmentZero32 must change segments to enable/disable isolation during function calls; second, it uses indirect function calls for invocation (a choice that simplifies engineering but is not fundamental), and finally, there is a small slowdown imposed by code structure enforced to allow zero-cost transitions.

Image rendering. The impact of the above sandboxing schemes on image rendering is compared in Figure 9 (standard deviations under 1%). For narrow images of width 10 pixels, SegmentZero32 overheads relative to the native unsandboxed code are 24%, 1%, and 6.5% for SimpleImage, StockImage and RandomImage, respectively. This is lower than the corresponding overheads for NaCl32 which are 312%, 29%, and 66% respectively as well as IdealHeavy32 which are 208%, 28% and 45% respectively. As observed in Wasm

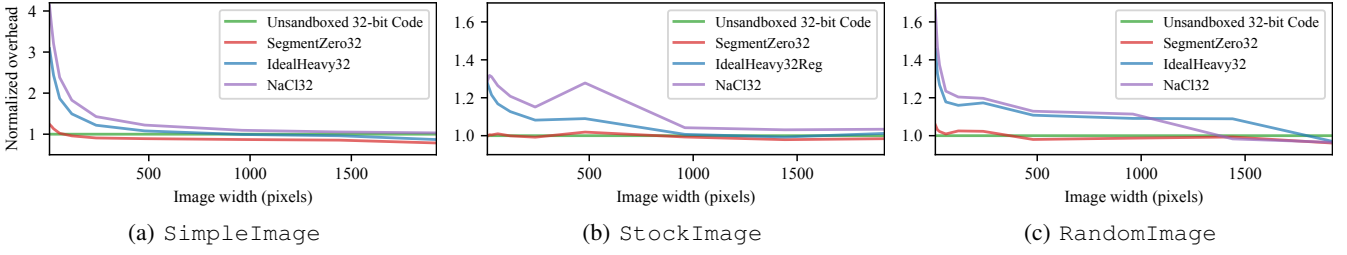


Figure 9: Performance of image rendering with libjpeg sandboxed with SegmentZero32 and NaCl32 and IdealHeavy32. Times are relative to unsandboxed code. NaCl32 and IdealHeavy32 relative overheads are as high as 312% and 208% respectively, while SegmentZero32 relative overheads do not exceed 24%.

workloads, these overheads reduce as image width increases and the complexity of the image increases. Additionally, we observe that the overheads are negligible for images wider than 480 pixels but, as we have seen earlier, such images constitute less than 20% of all images on the Alexa top 500 websites.

Conclusion. From our evaluation, we conclude that the performance of an isolation scheme that supports zero-cost transitions can be significantly lower for certain workloads, especially those with a large number of transitions. Additionally, for workloads that do not transition frequently (e.g., rendering wide images), the additional runtime overheads imposed by zero-cost condition enforcement do not result in significant performance overheads.

VII. RELATED WORK

A considerable amount of research has gone into efficient implementations of memory isolation and CFI techniques to provide SFI across many platforms [2], [3], [5]–[8], [11], [12], [14], [20]–[22], [38], [57]–[60]. However, these systems either implement or require the user to implement heavyweight springboards and trampolines to guarantee security.

SFI systems. Wahbe et al. [1] 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 [9] optimized transitions by clearing and saving contexts using machine specific mechanisms like the Intel® `fxrstor` instruction which clears floating point state and SIMD registers. We show (§VI) that such transitions still impose significant overhead. While CPU makers continue to add optimized context switching instructions, such instructions do not yet eliminate all overhead.

Zeng et al. [61] 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 [4] 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 [35] sandboxes C code through compilation to Wasm followed by (de)compilation back to C, ensuring that the sandboxed C 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. 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 Wasm to C.

Sledge [13] describes a Wasm runtime for edge computing, that relies on Wasm properties to enable efficient isolation of serverless components. However, Sledge focuses on function scheduling including preempting running Wasm programs, and its needs for context saving differ from library sandboxing as contexts must be saved even in the middle of function calls.

SFI Verification. Our work includes the compiler in the TCB. Previous work on SFI (e.g., [2], [4], [9], [62]) instead uses a *verifier* (a small verified trusted program) or a theorem prover [23], [63] to validate the relevant SFI properties of compiled sandbox code. However these verifiers do not currently establish sufficient properties for zero-cost transitions. Our safety monitor definition provides the appropriate context for extending existing SFI verification work to validate that compiled code meets both SFI and zero-cost conditions.

Hardware based isolation. Hardware features such as memory protection keys [25], [26], extended page tables [64], virtualization instructions [64], [65], or even dedicated hardware designs [66] 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 but require heavyweight transitions. IdealHeavy32 in Section VI studies an idealized version of such a scheme.

Capabilities. [67] and [68] both look at protecting interacting components on systems that provide hardware enforced capabilities. [67] 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 any formal security guarantees. [68] investigate a calling-convention based on capabilities such as those of CHERI [69] that allows safe sharing of a stack between distrusting components. Their definition of well-bracketed control flow and local state encapsulation via an overlay system 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 [32], [70], [71], 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) [33], [72], [73] 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.

VIII. DISCUSSION

Zero-cost transitions significantly simplify the transitions between application and sandbox and, as shown in Section VI, improve the performance of real workloads. Additionally, they may also offer benefits in performance and clarity in designing isolation schemes; we discuss these briefly.

Multi-threading and separate stacks. We believe that stack switching can incur larger overheads than shown in Section VI when running workloads where multiple application threads can invoke sandboxed code in parallel. Here, the application must maintain an n-to-n stack mapping, i.e., a corresponding sandbox stack for each application thread that invokes a sandboxed function. In large applications like browsers, it is not clear which threads invoke sandboxed code ahead of time; thus, existing tools [16] maintain and check the mapping lazily as part of the heavyweight trampoline, thereby adding additional latency to heavyweight transitions.

Alternate ABIs. We measure transition overhead with the System V 32-bit and 64-bit calling. However, other conventions such as the one used in Windows [74] requires saving and restoring 10 additional floating point registers and may incur large overhead. Further, heavyweight trampolines get more complicated and expensive with any ISA extensions that introduce more registers.

LTO. We show that zero-cost transitions applied to Wasm reduce transitions to a simple function call. This then allows LTO to optimize code across calls across the application-Wasm boundary. We don’t use LTO across the application sandbox boundary in our benchmarks, however LTO speedups for Wasm sandboxed code have been examined by Zakai [35].

Zero-cost transition compatible hardware. As we have demonstrated, hardware support for memory isolation can be used for zero-cost transitions if it can separate heap and stack operations. This simple observation provides a blueprint for how future isolation hardware (including any hardware extensions for 64-bit CPUs) could support zero-cost schemes.

REFERENCES

- [1] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, “Efficient software-based fault isolation,” in *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, ser. SOSP ’93. Association for Computing Machinery, 1993, pp. 203–216. [Online]. Available: <https://doi.org/10.1145/168619.168635>
- [2] S. McCamant and G. Morrisett, “Evaluating SFI for a CISC architecture,” 2006. [Online]. Available: <https://www.usenix.org/conference/15th-usenix-security-symposium/evaluating-sfi-cisc-architecture>
- [3] G. Tan et al., *Principles and implementation techniques of software-based fault isolation*. Now Publishers, 2017.
- [4] Ú. Erlingsson, M. Abadi, M. Vrabie, M. Budiu, and G. C. Necula, “XFI: Software guards for system address spaces,” in *OSDI*, 2006.
- [5] M. Castro, M. Costa, J.-P. Martin, M. Peinado, P. Akritidis, A. Donnelly, P. Barham, and R. Black, “Fast byte-granularity software fault isolation,” in *OSDI*, 2009.
- [6] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum, “Fault isolation for device drivers,” in *DSN*. IEEE, 2009.
- [7] M. I. Seltzer, Y. Endo, C. Small, and K. A. Smith, “Dealing with disaster: Surviving misbehaved kernel extensions,” in *OSDI*, vol. 96, no. 56, 1996.
- [8] S. Lucco, O. Sharp, and R. Wahbe, “Omnware: A universal substrate for web programming,” in *WWW*, 1995.
- [9] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, “Native client: A sandbox for portable, untrusted x86 native code,” in *2009 30th IEEE Symposium on Security and Privacy*, 2009, pp. 79–93, ISSN: 2375-1207.
- [10] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, “Bringing the web up to speed with WebAssembly,” in *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. Association for Computing Machinery, 2017, pp. 185–200. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=3062341.3062363>
- [11] J. Siefers, G. Tan, and G. Morrisett, “Robusta: Taming the native beast of the JVM,” in *CCS*, 2010.
- [12] B. Niu and G. Tan, “RockJIT: Securing just-in-time compilation using modular control-flow integrity,” in *CCS*, 2014.
- [13] P. K. Gadehalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmar, “Sledge: a serverless-first, light-weight wasm runtime for the edge,” in *Middleware*, 2020.
- [14] B. Ford and R. Cox, “Vx32: Lightweight user-level sandboxing on the x86,” in *USENIX ATC*, 2008.
- [15] B. Ford, “VXA: A virtual architecture for durable compressed archives,” in *FAST*, vol. 5, 2005.
- [16] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan, “Retrofitting fine grain isolation in the Firefox renderer,” in *USENIX Sec*, 2020.
- [17] 1566288 - rlbbox - port libgraphite usage code to use the rlbbox api. [Online]. Available: https://bugzilla.mozilla.org/show_bug.cgi?id=1566288
- [18] Pat Hickey, “Announcing Lucet: Fastly’s native WebAssembly compiler and runtime,” <https://www.fastly.com/blog/announcing-lucet-fastly-native-webassembly-compiler-runtime>, 2019.
- [19] T. McMullen, “Lucet: A compiler and runtime for high-concurrency low-latency sandboxing,” in *PriSC*, 2020.
- [20] M. Payer and T. R. Gross, “Fine-grained user-space security through virtualization,” 2011.

- [21] D. Sehr, R. Muth, K. Schimpf, C. Biffle, V. Khimenko, B. Yee, B. Chen, and E. Pasko, "Adapting software fault isolation to contemporary CPU architectures," in *USENIX Sec*, 2010.
- [22] A.-R. Adl-Tabatabai, G. Langdale, S. Lucco, and R. Wahbe, "Efficient and language-independent mobile programs," in *PLDI*, 1996.
- [23] L. Zhao, G. Li, B. De Sutter, and J. Regehr, "ARMor: fully verified software fault isolation," in *EMSOFT*, 2011.
- [24] Y. Zhou, X. Wang, Y. Chen, and Z. Wang, "Armlock: Hardware-based fault isolation for arm," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, 2014, pp. 558–569.
- [25] A. Vahldiek-Oberwagner, E. Elnikety, N. O. Duarte, M. Sammler, P. Druschel, and D. Garg, "ERIM: Secure, efficient in-process isolation with protection keys (MPK)," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*. {USENIX} Association, 2019, pp. 1221–1238.
- [26] M. Hedayati, S. Gravani, E. Johnson, J. Criswell, M. L. Scott, K. Shen, and M. Marty, "Hodor: Intra-process isolation for high-throughput data plane libraries," in *USENIX ATC*, 2019.
- [27] F. Alder, J. Van Bulck, D. Oswald, and F. Piessens, "Faulty Point Unit: ABI poisoning attacks on Intel SGX," in *ACSAC*, 2020.
- [28] S. Maffei, J. C. Mitchell, and A. Taly, "Object capabilities and isolation of untrusted web applications," in *IEEE S&P*, 2010.
- [29] A. Mettler, D. A. Wagner, and T. Close, "Joe-E: A security-oriented subset of Java," in *Network and Distributed System Security Symposium (NDSS)*, 2010.
- [30] M. Grimmer, R. Schatz, C. Seaton, T. Würthinger, and H. Mössenböck, "Memory-safe execution of C on a Java VM," in *Workshop on Programming Languages and Analysis for Security (PLAS)*, 2015.
- [31] M. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay, "Caja: Safe active content in sanitized javascript," <http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf>, June 2008.
- [32] G. C. Hunt and J. R. Larus, "Singularity: rethinking the software stack," *SIGOPS Operating Systems Review*, vol. 41, no. 2, 2007.
- [33] G. Morrisett, K. Crary, N. Glew, D. Grossman, R. Samuels, F. Smith, D. Walker, S. Weirich, and S. Zdancewic, "TALx86: A Realistic Typed Assembly Language," *ACM SIGPLAN Workshop on Compiler Support for System Software*, pp. 25–35, 1999.
- [34] J. Bosamiya, B. Lim, and B. Parno, "WebAssembly as an intermediate language for provably-safe software sandboxing," *PriSC*, 2020.
- [35] A. Zakai, "Wasmbbox: Simple, easy, and fast vm-less sandboxing," <https://kripken.github.io/blog/wasm/2020/07/27/wasmbbox.html>, 2020.
- [36] S. Narayan, T. Garfinkel, S. Lerner, H. Shacham, and D. Stefan, "Gobi: Webassembly as a practical path to library sandboxing," 2019.
- [37] H. Lu, M. Matz, M. Girkar, J. Hubička, A. Jaeger, and M. Mitchell, "System v application binary interface amd64 architecture processor supplement (with lp64 and ilp32 programming models)," Tech. Rep., 2018. [Online]. Available: <https://software.intel.com/content/dam/develop/external/us/en/documents/intro-to-intel-avx-183287.pdf>
- [38] Lucet. [Online]. Available: <https://github.com/bytecodealliance/lucet>
- [39] Webassembly micro runtime. [Online]. Available: <https://github.com/bytecodealliance/wasm-micro-runtime>
- [40] Native Client team, "Native Client security contest archive," <https://developer.chrome.com/docs/native-client/community/security-contest/>, 2009.
- [41] "Issue 2919: Security: Naclswitch() leaks naclthreadcontext pointer to x86-32 untrusted code," <https://bugs.chromium.org/p/nativeclient/issues/detail?id=2919>, 2012.
- [42] "Issue 775: Uninitialized sendmsg syscall arguments in sel_ldr," <https://bugs.chromium.org/p/nativeclient/issues/detail?id=775>, 2010.
- [43] "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>, 2011.
- [44] "Issue 1633: Inner sandbox escape on 64-bit windows via kuserexceptiondispatcher," <https://bugs.chromium.org/p/nativeclient/issues/detail?id=1633>, 2011.
- [45] A. Bartel and J. Doe, "Twenty years of escaping the Java sandbox," in *Phrack*, 2018.
- [46] A. Matos and G. Boudol, "On declassification and the non-disclosure policy," in *18th IEEE Computer Security Foundations Workshop (CSFW'05)*, 2005, pp. 226–240.
- [47] "Java platform, standard edition: Java virtual machine guide," Tech. Rep., 2019. [Online]. Available: <https://docs.oracle.com/en/java/javase/13/vm/java-virtual-machine-guide.pdf>
- [48] "Intel® 64 and IA-32 architectures software developer's manual," 2020.
- [49] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 147–163. [Online]. Available: <https://www.usenix.org/conference/osdi14/technical-sessions/presentation/kuznetsov>
- [50] Safestack, Clang 12 documentation. [Online]. Available: <https://clang.llvm.org/docs/SafeStack.html>
- [51] N. Burow, S. A. Carr, J. Nash, P. Larsen, M. Franz, S. Brunthaler, and M. Payer, "Control-Flow Integrity: Precision, Security, and Performance," *ACM Computing Surveys*, vol. 50, pp. 16:1–16:33, Apr. 2017. [Online]. Available: <https://doi.org/10.1145/3054924>
- [52] Control Flow Integrity, Clang 12 documentation. [Online]. Available: <https://clang.llvm.org/docs/ControlFlowIntegrity.html>
- [53] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, Ú. Erlingsson, L. Lozano, and G. Pike, "Enforcing forward-edge control-flow integrity in GCC & LLVM," in *Proceedings of the 23rd USENIX Security Symposium*, K. Fu and J. Jung, Eds., 2014, pp. 941–955.
- [54] Automatic variable initialization. [Online]. Available: <https://reviews.llvm.org/rL349442>
- [55] Shielding linux resources—introduction. [Online]. Available: <https://documentation.suse.com/sle-rt/15-SP1/html/SLE-RT-all/cha-shielding-intro.html>
- [56] L. T. Hansen, "Craneflight: Performance parity with Baldr on x86-64," https://bugzilla.mozilla.org/show_bug.cgi?id=1539399, 2019.
- [57] N. Goonasekera, W. Caelli, and C. Fidge, "LibVM: an architecture for shared library sandboxing," vol. 45, no. 12, pp. 1597–1617, 2015. [Online]. Available: <https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2294>
- [58] A. Bittau, P. Marchenko, M. Handley, and B. Karp, "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*, J. Crowcroft and M. Dahlin, Eds. USENIX Association, 2008, pp. 309–322. [Online]. Available: http://www.usenix.org/events/nsdi08/tech/full_papers/bittau/bittau.pdf
- [59] J. Litton, A. Vahldiek-Oberwagner, E. Elnikety, D. Garg, B. Bhattacharjee, and P. Druschel, "Light-weight contexts: An os abstraction for safety and performance," in *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'16. USENIX Association, 2016, p. 49–64.
- [60] Y. Chen, S. Reymondjohnson, Z. Sun, and L. Lu, "Shreds: Fine-grained execution units with private memory," in *2016 IEEE Symposium on Security and Privacy (SP)*, 2016, pp. 56–71.
- [61] B. Zeng, G. Tan, and G. Morrisett, "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*, ser. CCS '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 29–40. [Online]. Available: <https://doi.org/10.1145/2046707.2046713>
- [62] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Trust, but verify: SFI safety for native-compiled Wasm," in *Network and Distributed System Security Symposium (NDSS)*. Internet Society, February 2021.
- [63] J. A. Kroll, G. Stewart, and A. W. Appel, "Portable software fault isolation," in *2014 IEEE 27th Computer Security Foundations Symposium*. IEEE, 2014, pp. 18–32.
- [64] W. Qiang, Y. Cao, W. Dai, D. Zou, H. Jin, and B. Liu, "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)*, 2017, pp. 34–41.
- [65] A. Belay, A. Bittau, A. Mashtizadeh, D. Terei, D. Mazières, and C. Kozyrakis, "Dune: Safe user-level access to privileged cpu features," in *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'12. USA: USENIX Association, 2012, p. 335–348.
- [66] D. Schrammel, S. Weiser, S. Steinegger, M. Schwarzl, M. Schwarz, S. Mangard, and D. Gruss, "Donky: Domain keys – efficient in-process isolation for risc-v and x86," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1677–1694. [Online]. Available: <https://www.usenix.org/conference/usenixsecurity20/presentation/schrammel>

- [67] P. A. Karger, “Using registers to optimize cross-domain call performance,” in *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS III. New York, NY, USA: Association for Computing Machinery, 1989, p. 194–204. [Online]. Available: <https://doi.org/10.1145/70082.68201>
- [68] L. Skorstengaard, D. Devriese, and L. Birkedal, “StkTokens: enforcing well-bracketed control flow and stack encapsulation using linear capabilities,” *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–28, Jan. 2019. [Online]. Available: <http://dl.acm.org/citation.cfm?doid=3302515.3290332>
- [69] R. N. M. Watson, J. Woodruff, P. G. Neumann, S. W. Moore, J. Anderson, D. Chisnall, N. Dave, B. Davis, K. Gudka, B. Laurie, S. J. Murdoch, R. Norton, M. Roe, S. Son, and M. Vadera, “Cheri: A hybrid capability-system architecture for scalable software compartmentalization,” in *2015 IEEE Symposium on Security and Privacy*, 2015, pp. 20–37.
- [70] M. Aiken, M. Fähndrich, C. Hawblitzel, G. Hunt, and J. Larus, “Deconstructing process isolation,” in *Workshop on Memory system performance and correctness*, 2006.
- [71] M. Fähndrich, M. Aiken, C. Hawblitzel, O. Hodson, G. Hunt, J. R. Larus, and S. Levi, “Language support for fast and reliable message-based communication in Singularity OS,” in *EuroSys*. ACM, 2006.
- [72] G. Morrisett, D. Walker, K. Crary, and N. Glew, “From System F to Typed Assembly Language,” *ACM Transactions on Programming Languages and Systems*, vol. 21, pp. 527–568, May 1999. [Online]. Available: <https://doi.org/10.1145/319301.319345>
- [73] G. Morrisett, K. Crary, N. Glew, and D. Walker, “Stack-Based Typed Assembly Language,” *Journal of Functional Programming*, vol. 12, pp. 43–88, Jan. 2002, publisher: Cambridge University Press. [Online]. Available: <https://www.cambridge.org/core/journals/journal-of-functional-programming/article/abs/stackbased-typed-assembly-language/FAA86C307845C6E28B88F57EE64C6F3B>
- [74] “x64 calling convention,” Tech. Rep., 2020. [Online]. Available: <https://docs.microsoft.com/en-us/cpp/build/x64-calling-convention?view=msvc-160>

APPENDIX A LANGUAGE DEFINITIONS

Figure 10 presents the syntax of our sandbox language model. For all programs we define the regions M , M_p , H_p , S_p , C_p , and I . $M = \mathbb{N}$ and represents the whole memory space. M_p , H_p , and S_p are the memory, heap, and stack of the application or library where the heap and stack sit disjointly inside the memory. C_p is set of instruction indices such that $C(n) = (p, _)$. I is the set of import indices, the beginnings of application functions that the library is allowed to jump to.

A note on calling convention: arguments are passed on the stack and the return address is placed above the arguments. When the application passes arguments to the library it marks their integrity as `lib`.

	pc, sp, n, ℓ	\in	\mathbb{N}
$Priv$	\ni	p	$::= \text{app} \mid \text{lib}$
Val	\ni	v	$::= \langle n, p \rangle$
Reg	\ni	r	$::= r_n \mid sp \mid pc$
$Check$	\ni	k	$\in \mathbb{N} \rightarrow \mathbb{N}$
$Immediate$	\ni	i	$::= r \mid v \mid i \oplus i$
$Command$	\ni	c	$::= r \leftarrow \text{pop}_p$ $\mid \text{push}_p i$ $\mid r \leftarrow \text{load}_k i$ $\mid \text{store}_k i := i$ $\mid r \leftarrow \text{mov } i$ $\mid \text{call}_k i$ $\mid \text{ret}_k$ $\mid \text{jmp}_k i$ $\mid r \leftarrow \text{movlabel}_p$ $\mid \text{storelabel}_p i$ $\mid \text{gatecall}_n i$ $\mid \text{gateret}$
$Code$	\ni	C	$::= \mathbb{N} \rightarrow Priv \times Command$
$RegVals$	\ni	R	$::= Reg \rightarrow Val$
$Memory$	\ni	M	$::= \mathbb{N} \rightarrow Val$
$State$	\ni	Ψ	$::= \text{error}$ $\mid \{ \begin{array}{ll} pc & : \mathbb{N} \\ sp & : \mathbb{N} \\ R & : RegVals \\ M & : Memory \\ C & : Code \end{array} \}$

Figure 10: Syntax

Figure 11 and Figure 13 define the base small step operational semantics. We separate this into transitions $\Psi(\langle c \rangle) \rightarrow \Psi'$ and error transitions $\Psi(\langle c \rangle) \rightarrow \text{error}$.

$\Psi(\langle c \rangle) \rightarrow \Psi'$

$$\begin{array}{c}
\frac{\Psi.sp \in S_{p_s} \quad p_s \sqsubseteq p \quad v = \Psi.M(\Psi.sp) \quad R' = R[r \mapsto v]}{\Psi(r \leftarrow \text{pop}_p) \rightarrow \Psi^{++}[sp := \Psi.sp - 1, R := R']} \quad \frac{v = \mathcal{V}_\Psi(i) \quad sp' = \Psi.sp + 1 \quad M' = \Psi.M[sp' \mapsto v] \quad sp' \in S_{p_s} \quad p_s \sqsubseteq p}{\Psi(\text{push}_p i) \rightarrow \Psi^{++}[sp := sp', M := M']} \\
\\
\frac{\langle n \rangle = \mathcal{V}_\Psi(i) \quad n' = k(n) \quad v = \Psi.M(n') \quad R' = \Psi.R[r \mapsto v]}{\Psi(r \leftarrow \text{load}_k i) \rightarrow \Psi^{++}[R := R']} \quad \frac{\langle n \rangle = \mathcal{V}_\Psi(i) \quad v = \mathcal{V}_\Psi(i') \quad n' = k(n) \quad M' = \Psi.M[n' \mapsto v]}{\Psi(\text{store}_k i := i') \rightarrow \Psi^{++}[M := M']} \quad \frac{v = \mathcal{V}_\Psi(i) \quad R' = \Psi.R[r \mapsto v]}{\Psi(r \leftarrow \text{mov } i) \rightarrow \Psi^{++}[R := R']} \\
\\
\frac{M' = \Psi.M[n := \langle m, p \rangle] \quad \langle n \rangle = \mathcal{V}_\Psi(i) \quad \langle m \rangle = \Psi.M(n)}{\Psi(\text{storelabel}_p i) \rightarrow \Psi^{++}[M := M']} \quad \frac{\langle n \rangle = \mathcal{V}_\Psi(i) \quad n' = k(n) \quad sp' = \Psi.sp + 1 \quad M' = \Psi.M[sp' \mapsto \langle \Psi.pc + 1, \text{lib} \rangle] \quad sp' \in S_{p_s}}{\Psi(\text{call}_k i) \rightarrow \Psi[pc := n', sp := sp', M := M']} \\
\\
\frac{\langle n \rangle = \mathcal{V}_\Psi(i) \quad n' = k(n)}{\Psi(\text{jmp}_k i) \rightarrow \Psi[pc := n']} \quad \frac{\langle n \rangle = \Psi.M(\Psi.sp) \quad n' = k(n) \quad \Psi.sp \in S_{p_s}}{\Psi(\text{ret}_k) \rightarrow \Psi[pc := n', sp := \Psi.sp - 1]} \quad \frac{\langle n \rangle = \Psi.R(r) \quad R' = \Psi.R[r := \langle n, p \rangle]}{\Psi(r \leftarrow \text{movlabel}_p) \rightarrow \Psi^{++}[R := R']} \\
\\
\frac{\langle v \rangle = \mathcal{V}_\Psi(i)}{\Psi(sp \leftarrow \text{mov } i) \rightarrow \Psi^{++}[sp := v]}
\end{array}$$

Figure 11: Operational Semantics

$\Psi(\langle c \rangle) \rightarrow \text{error}$

$$\begin{array}{c}
\frac{\Psi.sp \in S_{p_s} \quad p_s \not\sqsubseteq p}{\Psi(r \leftarrow \text{pop}_p) \rightarrow \text{error}} \quad \frac{\Psi.sp + 1 \in S_{p_s} \quad p_s \not\sqsubseteq p}{\Psi(\text{push}_p i) \rightarrow \text{error}} \quad \frac{\langle n \rangle = \mathcal{V}_\Psi(i) \quad k(n) \text{ undefined}}{\Psi(r \leftarrow \text{load}_k i) \rightarrow \text{error}} \\
\\
\frac{\langle n \rangle = \mathcal{V}_\Psi(i) \quad k(n) \text{ undefined}}{\Psi(\text{store}_k i := i') \rightarrow \text{error}} \quad \frac{\Psi.sp + 1 \notin S_{p_s}}{\Psi(\text{call}_k i) \rightarrow \text{error}} \quad \frac{\langle n \rangle = \mathcal{V}_\Psi(i) \quad k(n) \text{ undefined}}{\Psi(\text{call}_k i) \rightarrow \text{error}} \quad \frac{\Psi.sp \notin S_{p_s}}{\Psi(\text{ret}_k) \rightarrow \text{error}} \\
\\
\frac{\langle n \rangle = \Psi.M(\Psi.sp) \quad k(n) \text{ undefined}}{\Psi(\text{ret}_k) \rightarrow \text{error}} \quad \frac{\langle n \rangle = \mathcal{V}_\Psi(i) \quad k(n) \text{ undefined}}{\Psi(\text{jmp}_k i) \rightarrow \text{error}}
\end{array}$$

Figure 12: Operational Semantics

$$\begin{array}{c}
\frac{C(\Psi.pc) = (_, c)}{\Psi(\langle c \rangle)} \quad \frac{C(\Psi.pc) = (p, c)}{\Psi(\langle c \rangle)_p} \quad \frac{\Psi \rightarrow^* \Psi' \quad \neg \exists \Psi''. \Psi' \rightarrow \Psi'' \quad \Psi' \neq \text{error}}{\Psi \Downarrow \Psi'} \quad \text{lib} \sqsubseteq \text{app} \\
\\
\begin{array}{l}
\mathcal{V}_\Psi(v) \triangleq v \\
\mathcal{V}_\Psi(r) \triangleq \Psi.R(r) \\
\mathcal{V}_\Psi(sp) \triangleq \langle \Psi.sp, \text{lib} \rangle \\
\mathcal{V}_\Psi(pc) \triangleq \langle \Psi.pc, \text{lib} \rangle \\
\mathcal{V}_\Psi(i \oplus i') \triangleq \langle v \oplus v', p \sqcap p' \rangle \\
\text{where } \langle v, p \rangle = \mathcal{V}_\Psi(i) \\
\quad \quad \langle v', p' \rangle = \mathcal{V}_\Psi(i')
\end{array} \\
\\
\begin{array}{l}
\langle n \rangle \triangleq \langle n, _ \rangle \\
\Psi^{++} \triangleq \Psi[pc := \Psi.pc + 1]
\end{array}
\end{array}$$

Figure 13: Operational Semantics: Auxiliary Definitions

Figure 14 defines unguarded derived forms for memory operations.

$$\begin{aligned}
r \leftarrow \text{pop} &\triangleq r \leftarrow \text{pop}_\top \\
\text{push } i &\triangleq \text{push}_\top i \\
r \leftarrow \text{load } i &\triangleq r \leftarrow \text{load}_{id} i \\
\text{store } i := i' &\triangleq \text{store}_{id} i := i' \\
\text{jmp } i &\triangleq \text{jmp}_{id} i \\
\text{call } i &\triangleq \text{call}_{id} i \\
\text{ret} &\triangleq \text{ret}_{id}
\end{aligned}$$

Figure 14: Derived Forms

A. Sandbox Properties

$$\begin{array}{c}
\frac{\Psi_1 \rightarrow \Psi_2 \quad \Psi_1 \langle c_1 \rangle_{p_1} \quad \Psi_2 \langle c_2 \rangle_{p_2} \quad p_1 = p_2 = p}{\Psi_1 \xrightarrow{p} \Psi_2} \quad \frac{\Psi \xrightarrow{p} \Psi'}{\Psi \xrightarrow{\square} \Psi'} \quad \frac{\Psi \xrightarrow{wb} \Psi'}{\Psi \xrightarrow{\square} \Psi'} \quad \frac{\Psi \rightarrow \Psi_1 \xrightarrow{\square}^* \Psi_2 \rightarrow \Psi' \quad \Psi \langle \text{gatecall}_n i \rangle \quad \Psi_2 \langle \text{gateret} \rangle}{\Psi \xrightarrow{wb} \Psi'}
\end{array}$$

Figure 15: Well-Bracketed Transitions

1) *Integrity*: Integrity is all about maintaining application invariants across calls into the sandbox. These invariants vary significantly from program to program, so to capture this generality we define \mathcal{I} -Integrity and then instantiate it in several specific instances.

Definition 5 (\mathcal{I} -Integrity).

Let $\mathcal{I} : \text{Trace} \times \text{State} \times \text{State} \rightarrow \mathbb{P}$. Then, if $\Psi_0 \in \text{Program}$, $\pi = \Psi_0 \rightarrow^* \Psi_1$, $\Psi_1.p = \text{app}$, and $\Psi_1 \xrightarrow{wb} \Psi_2$ imply that $\mathcal{I}(\pi, \Psi_1, \Psi_2)$, we say that an SFI transition system has \mathcal{I} -integrity.

Informally, callee-save register integrity says that the values of callee-save registers are restored by gated calls into the sandbox:

Definition 6 (Callee-Save Register Integrity).

Let CSR be the list of callee-save registers and define

$$\text{CSR}(_, \Psi_1, \Psi_2) \triangleq \Psi_1.R(\text{CSR}) = \Psi_2.R(\text{CSR}).$$

If an SFI transition system has CSR -integrity then we say the system has callee-save register integrity.

$$\begin{aligned}
&\text{return-address}_p : \text{Trace} \rightarrow \wp(\mathbb{N}) \\
&\text{return-address}_p(\Psi_0 \rightarrow^* \Psi \langle \text{call}_k i \rangle_p \rightarrow \Psi') \triangleq \text{return-address}_p(\Psi_0 \rightarrow^* \Psi) \cup \{\Psi.sp + 1\} \\
&\text{return-address}_p(\Psi_0 \rightarrow^* \Psi \langle \text{ret}_k \rangle_p \rightarrow \Psi') \triangleq \text{return-address}_p(\Psi_0 \rightarrow^* \Psi) - \{\Psi.sp\} \\
&\text{return-address}_p(\Psi_0 \rightarrow^* \Psi \langle \text{gatecall}_n i \rangle_p \rightarrow \Psi') \triangleq \text{return-address}_p(\Psi_0 \rightarrow^* \Psi) \cup \{\Psi.sp + 1\} \\
&\text{return-address}_p(\Psi_0 \rightarrow^* \Psi \langle \text{gateret} \rangle_p \rightarrow \Psi') \triangleq \text{return-address}_p(\Psi_0 \rightarrow^* \Psi) - \{\Psi.sp\} \\
&\text{return-address}_p(\Psi_0 \rightarrow^* \Psi \langle c \rangle \rightarrow \Psi') \triangleq \text{return-address}_p(\Psi_0 \rightarrow^* \Psi) \\
&\text{return-address}_p(\Psi_0 \rightarrow^0 \Psi_0) \triangleq \emptyset
\end{aligned}$$

Figure 16: Call stack return address calculation

Definition 7 (Return Address Integrity). Define

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

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

2) *Confidentiality*: Let V be a map from some type A to Val . We then define $V|_{lib}$ as the following restriction of V :

$$V|_{lib}(a) = \begin{cases} n & \text{when } V(a) = \langle n, lib \rangle \\ \text{undefined} & \text{otherwise} \end{cases}$$

We then say that $\Psi =_{lib} \Psi'$ if $\Psi.R|_{lib} = \Psi'.R|_{lib}$ and $\Psi.M|_{lib} = \Psi'.M|_{lib}$ where equality of partial functions requires that they have the same domain. We then define two notions of observational equivalence.

Definition 8. We say $\Psi =_{call\ n} \Psi'$ if

- 1) $\Psi.M(H_{lib}) = \Psi'.M(H_{lib})$
- 2) $\Psi.pc = \Psi'.pc$
- 3) $\Psi.sp = \Psi'.sp$
- 4) For all $i \in [1, n]$, there exists some n' such that $\langle n', lib \rangle = \Psi.M(\Psi.sp - i) = \Psi'.M(\Psi.sp - i)$.

Second, let r_{ret} be the calling convention return register.

Definition 9. We say $\Psi =_{ret} \Psi'$ if

- 1) $\Psi.M(H_{lib}) = \Psi'.M(H_{lib})$
- 2) $\Psi.pc = \Psi'.pc$
- 3) There exists some n such that $\langle n, lib \rangle = \Psi.R(r_{ret}) = \Psi'.R(r_{ret})$.

Definition 10 (Disjoint Noninterference).

If, for all $\Psi_0 \in Program$, $\Psi_1(\perp)_{lib}$, $\Psi_3(\perp)_{app}$, $\Psi_0 \rightarrow^* \Psi_1 \xrightarrow{lib}^n \Psi_2 \rightarrow \Psi_3$, and, for all Ψ'_1 such that $\Psi_1 =_{lib} \Psi'_1$, we have that $\Psi'_1 \xrightarrow{lib}^n \Psi'_2 \rightarrow \Psi'_3$, $\Psi'_3(\perp)_{app}$, $\Psi_3.pc = \Psi'_3.pc$, and

- 1) $\Psi_2(\text{gatecall}_{n'}\ i)$, $\Psi'_2(\text{gatecall}_{n'}\ i)$, and $\Psi_3 =_{call\ n'} \Psi'_3$ or
- 2) $\Psi_2(\text{gateret})$, $\Psi'_2(\text{gateret})$, and $\Psi_3 =_{ret} \Psi'_3$,

then we say that the SFI transition system has the disjoint noninterference property.

APPENDIX B	
NaCl	
NaCl context in application	
$ctx - 0$	library stack pointer
\dots	
ctx^*	ctx
NaCl context in library	
$ctx - 0$	application stack pointer
$ctx - 1$	\mathbb{CSR}_0
$ctx - 2$	\mathbb{CSR}_1
\dots	\dots
$ctx - \text{len}(\mathbb{CSR})$	$\mathbb{CSR}_{\text{len}(\mathbb{CSR})-1}$
\dots	
ctx^*	ctx

Figure 17: Transition Context Layout

$\text{nacl-springboard}(n, i) \triangleq$		
$j \in (\text{len}(\mathbb{CSR}), 0]$	$r_0 \leftarrow \text{load } ctx^*$	
	$r_1 \leftarrow \text{load } r_0$	// r_1 holds the library stack pointer
	$\text{store } r_0 := \mathbb{CSR}_j; r_0 \leftarrow r_0 + 1$	// save callee save registers
	$r_1 \leftarrow r_1 + n$	// set r_1 to the new top of the library stack
$j \in [0, n)$	$sp \leftarrow sp - 1$	// move the stack pointer to the first argument
	$r_2 \leftarrow \text{pop}; \text{store}_{M_{lib}}\ r_1 := r_2; r_1 \leftarrow r_1 - 1$	// copy arguments to library stack
	$r_2 \leftarrow sp + (n + 1); \text{store } r_0 := r_2$	// save stack pointer
	$sp \leftarrow r_1 + n$	// set new stack pointer
$r \in \mathbb{R}$	$\text{store } ctx^* := r_0$	// update ctx
	$r \leftarrow \text{mov } \langle 0, lib \rangle$	// clear registers
	$\text{jmp } i$	

$\text{nacl-trampoline} \triangleq$	
$j \in [0, \text{len}(\text{CSR}))$	$r_0 \leftarrow \text{load } ctx^*$ $r_0 \leftarrow r_0 - 1; \text{CSR}_j \leftarrow \text{load } r_0$ // restore callee save registers $r_0 \leftarrow \text{load } ctx^*$ $r_1 \leftarrow \text{load } r_0$ // r_1 holds the application stack pointer $r_0 \leftarrow r_0 - \text{len}(\text{CSR}); \text{store } r_0 := sp$ // save library stack pointer $\text{store } ctx^* := r_0$ // update ctx $sp \leftarrow \text{mov } r_1$ // switch to application stack ret
$\text{nacl-cb-springboard}(n, i) \triangleq$	
$j \in [0, n)$	$r_0 \leftarrow \text{load } ctx^*$ $r_1 \leftarrow \text{load } r_0$ // r_1 holds the application stack pointer $r_0 \leftarrow r_0 + 1; \text{store } r_0 := sp$ // save stack pointer $\text{store } ctx^* := r_0$ // update ctx $sp \leftarrow sp - 1$ // move the stack pointer to the first argument $r_1 \leftarrow r_1 + n$ // set r_1 to the new top of the library stack $r_2 \leftarrow \text{pop}_{M_{\text{lib}}}; \text{store } r_1 := r_2; r_1 \leftarrow r_1 - 1$ // copy arguments to application stack $sp \leftarrow r_1 + n$ // set new stack pointer $\text{jmp}_I i$
$\text{nacl-cb-trampoline} \triangleq$	
$r \in \mathbb{R}$	$r_0 \leftarrow \text{load } ctx^*$ $r_1 \leftarrow \text{load } r_0$ // r_1 holds the library stack pointer $r_0 \leftarrow r_0 - 1; \text{store } ctx^* := r_0$ // update ctx $sp \leftarrow \text{mov } r_1$ // switch to library stack $r \leftarrow \text{mov } \langle 0, \text{lib} \rangle$ // clear registers $\text{ret}_{C_{\text{lib}}}$

A. Programs

A NaCl program Ψ is defined by the following conditions:

- 1) All memory operations in the sandboxed library are guarded:

$$\begin{aligned}
\forall n. \Psi.C(n) = (\text{lib}, r \leftarrow \text{pop}_p) &\implies p = \text{lib} \\
\Psi.C(n) = (\text{lib}, \text{push}_p i) &\implies p = \text{lib} \\
\Psi.C(n) = (\text{lib}, r \leftarrow \text{load}_{chk} i) &\implies chk \subseteq M_{\text{lib}} \\
\Psi.C(n) = (\text{lib}, \text{store}_{chk} i := i) &\implies chk \subseteq M_{\text{lib}}.
\end{aligned}$$

- 2) The application does not write app data to the sandbox memory.
- 3) Gated calls are the only way to move between application and library code:

$$\begin{aligned}
\forall n. \Psi.C(n) = (p, \text{call}_{chk} i) &\implies chk \subseteq C_p \\
\Psi.C(n) = (p, \text{ret}_{chk}) &\implies chk \subseteq C_p \\
\Psi.C(n) = (p, \text{jmp}_{chk} i) &\implies chk \subseteq C_p
\end{aligned}$$

- 4) The sandboxed library cannot change integrity labels:

$$\begin{aligned}
\forall n. \Psi.C(n) &\neq (\text{lib}, r \leftarrow \text{movlabel}_p) \\
\Psi.C(n) &\neq (\text{lib}, \text{storelabel}_p i).
\end{aligned}$$

- 5) The program starts in the application: $\Psi.pc = 0$ and $\Psi.C(0) = (\text{app}, _)$.
- 6) ctx^* and ctx start initialized to the library stack:

$$\begin{aligned}
ctx^* &\in H_{\text{app}} \\
\langle ctx \rangle &= \Psi.M(ctx^*) \in H_{\text{app}} \\
\Psi.M(ctx) &= S_{\text{lib}}[0] - 1.
\end{aligned}$$

B. Properties

Throughout the following we will use the shorthand $ctx_\Psi \triangleq \Psi.M(ctx^*)$.

Proposition 1. *NaCl has the disjoint noninterference property.*

Proof. Follows immediately from the fact that all reads and writes are guarded to be within M_{lib} , all values in M_{lib} have label `lib`, and all jumps remain within the library code. \square

Lemma 1. *The trampoline context is in H_{app} .*

Lemma 2. $ctx \geq ctx_0$.

Lemma 3. *If $\Psi_1 \langle c \rangle_{lib}$ and $\Psi_1 \xrightarrow{p}^* \Psi_2$, then $\Psi'' . M_{app} = \Psi' . M_{app}$.*

Proof. There are two cases for p : $p = \text{app}$ and $p = \text{lib}$. If $p = \text{app}$, then $\Psi_2 = \Psi_1$ and therefore trivially $\Psi_2 . M_{app} = \Psi_1 . M_{app}$. If $p = \text{lib}$, then for all Ψ such that $\Psi_1 \rightarrow^* \Psi \rightarrow^+ \Psi_2$, $\Psi.C(\Psi.pc) = (\text{lib}, _)$. By the structure of a NaCl program, this ensures that $\Psi_2 . M_{app} = \Psi_1 . M_{app}$. \square

Lemma 4. *If $\Psi \xrightarrow{p}^* \Psi'$ then $\Psi.p = \Psi'.p$.*

Lemma 5. *If $\Psi \xrightarrow{wb} \Psi'$ where $\Psi \rightarrow^* \Psi'' \rightarrow \Psi'$, then $\Psi'' . p \neq \Psi.p$ and $\Psi'.p = \Psi.p$.*

Proof. We proceed by simultaneous induction on the well-bracketed transition $\Psi \xrightarrow{wb} \Psi'$ and the length of $\Psi_0 \rightarrow^* \Psi \xrightarrow{wb} \Psi'$.

Case No callbacks

We have that $\Psi \langle \text{gatecall}_n \ i \rangle$ and there exist Ψ_1, Ψ_2 , and p such that $\Psi \rightarrow \Psi_1 \xrightarrow{p}^* \Psi_2 \rightarrow \Psi'$ where $\Psi_2 \langle \text{gateret} \rangle$. Here $\Psi'' = \Psi_2$. By inspection of the reduction for $\text{gatecall}_n \ i$ we know that $\Psi_1.p \neq \Psi.p$ and therefore by Lemma 4 $\Psi'' . p \neq \Psi.p$. By inspection of the reduction for gateret , $\Psi'.p = \Psi.p$.

Case Callbacks

We have that $\Psi \langle \text{gatecall}_n \ i \rangle$ and there exist Ψ_1, Ψ_2 such that $\Psi \rightarrow \Psi_1 \xrightarrow{\square}^* \Psi_2 \rightarrow \Psi'$ where $\Psi_2 \langle \text{gateret} \rangle$. Here $\Psi'' = \Psi_2$. By inspection of the reduction for $\text{gatecall}_n \ i$ we know that $\Psi_1.p \neq \Psi.p$. We now show, by induction on $\Psi_1 \xrightarrow{\square}^* \Psi_2$, that $\Psi_2.p = \Psi_1.p \neq \Psi.p$.

Proof. If there are no steps then clearly $\Psi_2.p = \Psi_1.p \neq \Psi.p$. There are two possible cases for $\Psi_1 \xrightarrow{\square}^* \Psi_3 \xrightarrow{\square} \Psi_4$. When $\Psi_3 \xrightarrow{p} \Psi_4$, Lemma 4 gives us that $\Psi_4.p = \Psi_3.p = \Psi_1.p \neq \Psi.p$. When $\Psi_3 \xrightarrow{wb} \Psi_4$, our outer inductive hypothesis gives us that $\Psi_4.p = \Psi_3.p = \Psi_1.p \neq \Psi.p$. \blacksquare

Lastly, by inspection of the reduction for gateret , $\Psi'.p = \Psi.p$. \square

Lemma 6. *If $\Psi \xrightarrow{\square}^* \Psi'$, then $\Psi'.p = \Psi.p$.*

Proof. By induction, Lemma 4, and Lemma 5. \square

Lemma 7 (Context Integrity). *Let $\Psi_0 \in \text{Program}$, $\Psi_0 \rightarrow^* \Psi$, and $\Psi \xrightarrow{wb} \Psi'$. Then*

- 1) *if $\Psi.p = \text{app}$, then $\Psi.M([ctx_{\Psi_0}, ctx_\Psi]) = \Psi'.M([ctx_{\Psi_0}, ctx_{\Psi'}])$ and $ctx_\Psi = ctx_{\Psi'}$,*
- 2) *if $\Psi.p = \text{lib}$, then $\Psi.M([ctx_{\Psi_0}, ctx_\Psi]) = \Psi'.M([ctx_{\Psi_0}, ctx_{\Psi'}])$ and $ctx_\Psi = ctx_{\Psi'}$.*

Proof. We proceed by mutual, simultaneous induction on the well-bracketed transition $\Psi_1 \xrightarrow{wb} \Psi_2$ and the length of $\Psi_0 \rightarrow^* \Psi \xrightarrow{wb} \Psi'$.

First we consider the case where $\Psi \langle c \rangle_{app}$.

Case No callbacks

We have that $\Psi \langle \text{gatecall}_n \ i \rangle$ and there exist Ψ_1, Ψ_2 , and p such that $\Psi \rightarrow \Psi_1 \xrightarrow{p}^* \Psi_2 \rightarrow \Psi'$ where $\Psi_2 \langle \text{gateret} \rangle$. By Lemma 3 we have that $\Psi_1 . M_{app} = \Psi_2 . M_{app}$. By assumption we have that $\Psi \langle \text{gatecall}_n \ i \rangle_{app}$ and therefore by Lemma 5 we have that $\Psi_2 \langle \text{gateret} \rangle_{lib}$. By Lemma 1, Lemma 2, and inspection of the reduction rules for $\text{gatecall}_n \ i$ and gateret we have that $\Psi.M([ctx_{\Psi_0}, ctx_\Psi]) = \Psi'.M([ctx_{\Psi_0}, ctx_{\Psi'}])$ and $ctx_\Psi = ctx_{\Psi'}$.

Case Callbacks

We have that $\Psi \langle \text{gatecall}_n \ i \rangle$ and there exist Ψ_1, Ψ_2 such that $\Psi \rightarrow \Psi_1 \xrightarrow{\square}^* \Psi_2 \rightarrow \Psi'$ where $\Psi_2 \langle \text{gateret} \rangle$. By inspection of the reduction rule for $\text{gatecall}_n \ i$ we have that $ctx_{\Psi_1} = ctx_\Psi + \text{len}(\mathbb{CS}\mathbb{R})$. We now show, by induction on $\Psi_1 \xrightarrow{\square}^* \Psi_2$, that $ctx_{\Psi_2} = ctx_{\Psi_1}$ and $\Psi_1.M([ctx_{\Psi_0}, ctx_{\Psi_1}]) = \Psi_2.M([ctx_{\Psi_0}, ctx_{\Psi_2}])$.

Proof. If there are no steps then clearly $ctx_{\Psi_2} = ctx_{\Psi_1}$ and all of $\Psi_1 . M_{app} = \Psi_2 . M_{app}$. There are two possible cases for $\Psi_1 \xrightarrow{\square}^* \Psi_3 \xrightarrow{\square} \Psi_4$, and notice that in both $\Psi_3.p = \Psi_1.p = \text{lib}$ (by Lemma 6). When $\Psi_3 \xrightarrow{p} \Psi_4$, Lemma 3 gives

us that $\Psi_3.M_{\text{app}} = \Psi_4.M_{\text{app}}$ and then Lemma 1 gives us that $\text{ctx}_{\Psi_4} = \text{ctx}_{\Psi_3} = \text{ctx}_{\Psi_1}$. When $\Psi_3 \xrightarrow{\text{wb}} \Psi_4$, case 2 of our inductive hypothesis gives us that $\Psi_1.M([\text{ctx}_{\Psi_0}, \text{ctx}_{\Psi_1}]) = \Psi_3.M([\text{ctx}_{\Psi_0}, \text{ctx}_{\Psi_3}]) = \Psi_4.M([\text{ctx}_{\Psi_0}, \text{ctx}_{\Psi_4}])$ and $\text{ctx}_{\Psi_4} = \text{ctx}_{\Psi_3} = \text{ctx}_{\Psi_1}$. ■

Finally, by Lemma 6 we get that $\Psi_2 = \text{lib}$ and then inspection of the reduction rule for `gateret` gives us that $\Psi.M([\text{ctx}_{\Psi_0}, \text{ctx}_{\Psi}]) = \Psi'.M([\text{ctx}_{\Psi_0}, \text{ctx}_{\Psi'}])$ and $\text{ctx}_{\Psi} = \text{ctx}_{\Psi'}$.

Second we consider the case where $\Psi(c)_{\text{lib}}$.

Case No callbacks

We have that $\Psi(\text{gatecall}_n i)$ and there exist Ψ_1, Ψ_2 , and p such that $\Psi \rightarrow \Psi_1 \xrightarrow{p} \Psi_2 \rightarrow \Psi'$ where $\Psi_2(\text{gateret})$. By the structure of a NaCl program we have that $\text{ctx}_{\Psi_1} = \text{ctx}_{\Psi_2}$ and $\Psi_1.M([\text{ctx}_{\Psi_0}, \text{ctx}_{\Psi_1}]) = \Psi_2.M([\text{ctx}_{\Psi_0}, \text{ctx}_{\Psi_2}])$. By assumption we have that $\Psi(\text{gatecall}_n i)_{\text{lib}}$ and therefore by Lemma 5 we have that $\Psi_2(\text{gateret})_{\text{app}}$. By Lemma 2 and inspection of the reduction rules for `gatecalln i` and `gateret` we have that $\Psi.M([\text{ctx}_{\Psi_0}, \text{ctx}_{\Psi}]) = \Psi'.M([\text{ctx}_{\Psi_0}, \text{ctx}_{\Psi'}])$ and $\text{ctx}_{\Psi} = \text{ctx}_{\Psi'}$.

Case Callbacks

We have that $\Psi(\text{gatecall}_n i)$ and there exist Ψ_1, Ψ_2 such that $\Psi \rightarrow \Psi_1 \xrightarrow{\square} \Psi_2 \rightarrow \Psi'$ where $\Psi_2(\text{gateret})$. By inspection of the reduction rule for `gatecalln i` we have that $\text{ctx}_{\Psi_1} = \text{ctx}_{\Psi} + 1$. We now show, by induction on $\Psi_1 \xrightarrow{\square} \Psi_2$, that $\text{ctx}_{\Psi_2} = \text{ctx}_{\Psi_1}$ and $\Psi_1.M([\text{ctx}_{\Psi_0}, \text{ctx}_{\Psi_1}]) = \Psi_2.M([\text{ctx}_{\Psi_0}, \text{ctx}_{\Psi_2}])$.

Proof. If there are no steps then clearly $\text{ctx}_{\Psi_2} = \text{ctx}_{\Psi_1}$ and all of $\Psi_1.M_{\text{app}} = \Psi_2.M_{\text{app}}$. There are two possible cases for $\Psi_1 \xrightarrow{\square} \Psi_3 \xrightarrow{\square} \Psi_4$, and notice that in both $\Psi_3.p = \Psi_1.p = \text{app}$ (by Lemma 6). When $\Psi_3 \xrightarrow{p} \Psi_4$, the structure of a NaCl program gives us that $\text{ctx}_{\Psi_1} = \text{ctx}_{\Psi_2}$ and $\Psi_1.M([\text{ctx}_{\Psi_0}, \text{ctx}_{\Psi_1}]) = \Psi_2.M([\text{ctx}_{\Psi_0}, \text{ctx}_{\Psi_2}])$. When $\Psi_3 \xrightarrow{\text{wb}} \Psi_4$, case 1 of our inductive hypothesis gives us that $\Psi_1.M([\text{ctx}_{\Psi_0}, \text{ctx}_{\Psi_1}]) = \Psi_3.M([\text{ctx}_{\Psi_0}, \text{ctx}_{\Psi_3}]) = \Psi_4.M([\text{ctx}_{\Psi_0}, \text{ctx}_{\Psi_4}])$ and $\text{ctx}_{\Psi_4} = \text{ctx}_{\Psi_3} = \text{ctx}_{\Psi_1}$. ■

Finally, by Lemma 6 we get that $\Psi_2 = \text{app}$ and then inspection of the reduction rule for `gateret` gives us that $\Psi.M([\text{ctx}_{\Psi_0}, \text{ctx}_{\Psi}]) = \Psi'.M([\text{ctx}_{\Psi_0}, \text{ctx}_{\Psi'}])$ and $\text{ctx}_{\Psi} = \text{ctx}_{\Psi'}$. □

Proposition 2. *NaCl has callee-save register integrity.*

Proof. Follows from Lemma 7, Lemma 5, and inspection of the reduction rules for `gatecalln i` and `gateret`. □

Lemma 8. *Let $\Psi_0 \in \text{Program}$, $\pi = \Psi_0 \rightarrow^* \Psi$, and $\Psi \xrightarrow{\text{wb}} \Psi'$, then $\Psi'.M(\text{return-address}_{\text{app}}(\pi)) = \Psi.M(\text{return-address}_{\text{app}}(\pi))$.*

Proof. First we consider the case where $\Psi(c)_{\text{app}}$.

Case No callbacks

We have that $\Psi(\text{gatecall}_n i)$ and there exist Ψ_1, Ψ_2 , and p such that $\Psi \rightarrow \Psi_1 \xrightarrow{p} \Psi_2 \rightarrow \Psi'$ where $\Psi_2(\text{gateret})$. By inspection of the reduction rule for `gatecalln i` we see that $\Psi_1.M(\Psi.sp + 1) = \Psi.pc + 1$. This is adding to the top of the stack, so by the structure of a NaCl program we have that $\Psi_1.M(\text{return-address}_{\text{app}}(\pi)) = \Psi.M(\text{return-address}_{\text{app}}(\pi))$. By Lemma 3 we have that $\Psi_1.M_{\text{app}} = \Psi_2.M_{\text{app}}$ and therefore Lemma 1 gives us that $\text{ctx}_{\Psi_2} = \text{ctx}_{\Psi_1}$ and $\Psi_2.M(\text{return-address}_{\text{app}}(\pi)) = \Psi_1.M(\text{return-address}_{\text{app}}(\pi))$. If we inspect the trampoline code we see that, right before we execute the `ret`, we have set sp to $\Psi_2.M(\text{ctx}_{\Psi_2}) = \Psi_1.M(\text{ctx}_{\Psi_1}) = \Psi.sp + 1$. Thus, after returning the only part of the application stack that we modify is $\Psi.sp + 1$. This, and the fact that $\Psi_2.M(\text{return-address}_{\text{app}}(\pi)) = \Psi.M(\text{return-address}_{\text{app}}(\pi))$ gives us that $\Psi.M(\text{return-address}_{\text{app}}(\pi)) = \Psi'.M(\text{return-address}_{\text{app}}(\pi))$.

Case Callbacks

We have that $\Psi(\text{gatecall}_n i)$ and there exist Ψ_1, Ψ_2 such that $\Psi \rightarrow \Psi_1 \xrightarrow{\square} \Psi_2 \rightarrow \Psi'$ where $\Psi_2(\text{gateret})$. By inspection of the reduction rule for `gatecalln i` we see that $\Psi_1.M(\Psi.sp + 1) = \Psi.pc + 1$. This is adding to the top of the stack, so by the structure of a NaCl program we have that $\Psi_1.M(\text{return-address}_{\text{app}}(\pi)) = \Psi.M(\text{return-address}_{\text{app}}(\pi))$. We now show, by induction on $\Psi_1 \xrightarrow{\square} \Psi_2$ that $\Psi_2.M(\text{ctx}_{\Psi_2}) = \Psi.sp + 1$ and $\Psi_2.M(\text{return-address}_{\text{app}}(\pi)) = \Psi_1.M(\text{return-address}_{\text{app}}(\pi))$.

Proof. If there are no steps then $\Psi_2 = \Psi_1$ and both goals hold immediately. There are two possible cases for $\Psi_1 \xrightarrow{\square} \Psi_3 \xrightarrow{\square} \Psi_4$ and notice that in both $\Psi_3.p = \Psi_1.p = \text{lib}$ (by Lemma 6). If $\Psi_3 \xrightarrow{p} \Psi_4$ then Lemma 3 gives us that $\Psi_4.M_{\text{app}} = \Psi_3.M_{\text{app}}$ and our goal holds (as all of $\text{return-address}_{\text{app}}(\pi)$ is in S_{app}). If $\Psi_3 \xrightarrow{\text{wb}} \Psi_4$, then Lemma 7 gives us that $\text{ctx}_{\Psi_3} = \text{ctx}_{\Psi_4}$ and $\Psi_3.M([\text{ctx}_{\Psi_0}, \text{ctx}_{\Psi_3}]) = \Psi_4.M([\text{ctx}_{\Psi_0}, \text{ctx}_{\Psi_4}])$ and therefore that $\Psi_4.M(\text{ctx}_{\Psi_4}) = \Psi.sp + 1$. $\text{return-address}_{\text{app}}(\Psi_0 \rightarrow^* \Psi_3) = \text{return-address}_{\text{app}}(\pi) \uplus \Psi.sp + 1$, so our inductive hypothesis gives us that $\Psi_4.M(\text{return-address}_{\text{app}}(\pi)) = \Psi_3.M(\text{return-address}_{\text{app}}(\pi))$. ■

Second we consider the case where $\Psi(c)_{\text{lib}}$.

Case No callbacks

We have that $\Psi(\text{gatecall}_n\ i)$ and there exist Ψ_1, Ψ_2 , and p such that $\Psi \rightarrow \Psi_1 \xrightarrow{p} \Psi_2 \rightarrow \Psi'$ where $\Psi_2(\text{gateret})$. By inspection of the reduction rule for $\text{gatecall}_n\ i$ we see that $\Psi_1.M(\text{return-address}_{\text{app}}(\pi)) = \Psi.M(\text{return-address}_{\text{app}}(\pi))$. By the structure of a NaCl program we have that any call stack elements that are added during the callback will be popped before the gateret . Thus, $\Psi_2.M(\text{return-address}_{\text{app}}(\pi)) = \Psi_1.M(\text{return-address}_{\text{app}}(\pi))$. Inspection of the reduction rule for gateret then gives us that $\Psi'.M(\text{return-address}_{\text{app}}(\pi)) = \Psi_2.M(\text{return-address}_{\text{app}}(\pi)) = \Psi.M(\text{return-address}_{\text{app}}(\pi))$.

Case Callbacks

We have that $\Psi(\text{gatecall}_n\ i)$ and there exist Ψ_1, Ψ_2 such that $\Psi \rightarrow \Psi_1 \xrightarrow{\square} \Psi_2 \rightarrow \Psi'$ where $\Psi_2(\text{gateret})$. By inspection of the reduction rule for $\text{gatecall}_n\ i$ we see that $\Psi_1.M(\text{return-address}_{\text{app}}(\pi)) = \Psi.M(\text{return-address}_{\text{app}}(\pi))$. We now show, by induction on $\Psi_1 \xrightarrow{\square} \Psi_2$ that $\Psi_2.M(\text{return-address}_{\text{app}}(\pi)) = \Psi_1.M(\text{return-address}_{\text{app}}(\pi))$.

Proof. If there are no steps then $\Psi_2 = \Psi_1$ and the goal holds immediately. There are two possible cases for $\Psi_1 \xrightarrow{\square} \Psi_3 \xrightarrow{\square} \Psi_4$ and notice that in both $\Psi_3.p = \Psi_1.p = \text{app}$ (by Lemma 6). If $\Psi_3 \xrightarrow{p} \Psi_4$ then the structure of a NaCl program gives us that any call stack elements that are added during the callback will be popped before the gateret , and therefore our inductive invariant is maintained. If $\Psi_3 \xrightarrow{\text{wb}} \Psi_4$, then notice that $\text{return-address}_{\text{app}}(\Psi_0 \rightarrow^* \Psi_3) = \text{return-address}_{\text{app}}(\pi)$, so our inductive hypothesis gives us that $\Psi_4.M(\text{return-address}_{\text{app}}(\pi)) = \Psi_3.M(\text{return-address}_{\text{app}}(\pi))$. ■

Inspection of the reduction rule for gateret then gives us that $\Psi'.M(\text{return-address}_{\text{app}}(\pi)) = \Psi_2.M(\text{return-address}_{\text{app}}(\pi)) = \Psi.M(\text{return-address}_{\text{app}}(\pi))$. □

Proposition 3. NaCl has return address integrity.

Proof. We have that $\Psi_0 \in \text{Program}$, $\pi = \Psi_0 \rightarrow^* \Psi$, $\Psi.p = \text{app}$, and $\Psi \xrightarrow{\text{wb}} \Psi'$ and wish to show that $\Psi.M(\text{return-address}_{\text{app}}(\pi)) = \Psi'.M(\text{return-address}_{\text{app}}(\pi))$, $\Psi'.sp = \Psi.sp$, and $\Psi'.pc = \Psi.pc + 1$.

Lemma 8 gives us that $\Psi.M(\text{return-address}_{\text{app}}(\pi)) = \Psi'.M(\text{return-address}_{\text{app}}(\pi))$. We proceed by simultaneous induction on the well-bracketed transition $\Psi \xrightarrow{\text{wb}} \Psi'$ and the length of $\Psi_0 \rightarrow^* \Psi \xrightarrow{\text{wb}} \Psi'$ to show that $\Psi'.sp = \Psi.sp$, and $\Psi'.pc = \Psi.pc + 1$.

Case No callbacks

We have that $\Psi(\text{gatecall}_n\ i)$ and there exist Ψ_1, Ψ_2 , and p such that $\Psi \rightarrow \Psi_1 \xrightarrow{p} \Psi_2 \rightarrow \Psi'$ where $\Psi_2(\text{gateret})$. By inspection of the reduction rule for $\text{gatecall}_n\ i$ we see that $\Psi_1.M(\Psi.sp + 1) = \Psi.pc + 1$ and $\Psi_1.M(\text{ctx}_{\Psi_1}) = \Psi.sp + 1$. By Lemma 3 we have that $\Psi_1.M_{\text{app}} = \Psi_2.M_{\text{app}}$ and therefore Lemma 1 gives us that $\text{ctx}_{\Psi_2} = \text{ctx}_{\Psi_1}$. If we inspect the trampoline code we see that, right before we execute the ret , we have set sp to $\Psi_2.M(\text{ctx}_{\Psi_2}) = \Psi_1.M(\text{ctx}_{\Psi_1}) = \Psi.sp + 1$. Thus, after returning we have that $\Psi'.pc = \Psi.pc + 1$, $\Psi'.sp = \Psi.sp$.

Case Callbacks

We have that $\Psi(\text{gatecall}_n\ i)$ and there exist Ψ_1, Ψ_2 such that $\Psi \rightarrow \Psi_1 \xrightarrow{\square} \Psi_2 \rightarrow \Psi'$ where $\Psi_2(\text{gateret})$. By inspection of the reduction rule for $\text{gatecall}_n\ i$ we see that $\Psi_1.M(\Psi.sp + 1) = \Psi.pc + 1$ and $\Psi_1.M(\text{ctx}_{\Psi_1}) = \Psi.sp + 1$. We now show, by induction on $\Psi_1 \xrightarrow{\square} \Psi_2$ that $\Psi_2.M(\Psi.sp + 1) = \Psi.pc + 1$ and $\Psi_2.M(\text{ctx}_{\Psi_2}) = \Psi.sp + 1$.

Proof. If there are no steps then $\Psi_2 = \Psi_1$ and both hold immediately. There are two possible cases for $\Psi_1 \xrightarrow{\square} \Psi_3 \xrightarrow{\square} \Psi_4$ and notice that in both $\Psi_3.p = \Psi_1.p = \text{lib}$ (by Lemma 6). If $\Psi_3 \xrightarrow{p} \Psi_4$ then Lemma 3 gives us that $\Psi_4.M_{\text{app}} = \Psi_3.M_{\text{app}}$ and both hold (as the invariants are on M_{app}). If $\Psi_3 \xrightarrow{\text{wb}} \Psi_4$, then Lemma 7 gives us that $\text{ctx}_{\Psi_3} = \text{ctx}_{\Psi_4}$ and $\Psi_3.M([\text{ctx}_{\Psi_0}, \text{ctx}_{\Psi_3}]) = \Psi_4.M([\text{ctx}_{\Psi_0}, \text{ctx}_{\Psi_4}])$ and therefore that $\Psi_4.M(\text{ctx}_{\Psi_4}) = \Psi.sp + 1$. $\text{return-address}_{\text{app}}(\Psi_0 \rightarrow^* \Psi_3) = \text{return-address}_{\text{app}}(\pi) \uplus \Psi.sp + 1$ so Lemma 8 gives us that $\Psi_4.M(\Psi.sp + 1) = \Psi_3.M(\Psi.sp + 1) = \Psi.pc + 1$. ■

Finally, if we inspect the trampoline code we see that, right before we execute the ret , we have set sp to $\Psi_2.M(\text{ctx}_{\Psi_2}) = \Psi.sp + 1$. Thus, after returning we have that $\Psi'.pc = \Psi.pc + 1$, $\Psi'.sp = \Psi.sp$. □

APPENDIX C OVERLAY SEMANTICS

Figure 19, Figure 20, and Figure 21 define the overlay monitor operational semantics.

$$\begin{array}{lcl}
\text{Frame} & \ni & \textcolor{blue}{SF} ::= \{ \begin{array}{l} \text{base} : \mathbb{N} \\ \text{ret-addr-loc} : \mathbb{N} \\ \text{csr-vals} : \wp(\text{Reg} \times \mathbb{N}) \end{array} \} \\
\text{Function} & \ni & \textcolor{blue}{F} ::= \{ \begin{array}{l} \text{instrs} : \mathbb{N} \rightarrow \text{Command} \\ \text{entry} : \mathbb{N} \\ \text{type} : \mathbb{N} \end{array} \} \\
\text{oState} & \ni & \textcolor{blue}{\Phi} ::= \text{oerror} \\
& & | \{ \begin{array}{l} \Psi : \text{State} \\ \text{funcs} : \mathbb{N} \rightarrow \text{Function} \\ \text{stack} : [\text{Frame}] \end{array} \}
\end{array}$$

Figure 18: Overlay Extended Syntax

$$\boxed{\Psi \langle c \rangle \rightsquigarrow \Psi'}$$

$$\begin{array}{c}
\frac{\langle n, p_i \rangle = \mathcal{V}_{\Phi}(i) \quad n' = k(n) \quad sp' = \Phi.sp + 1 \quad M' = \Phi.M[sp' \mapsto \Phi.pc + 1] \quad \text{typechecks}(\Phi, n', sp') \quad \textcolor{blue}{SF} = \text{new-frame}(\Phi, n', sp') \quad p_i \sqsubseteq \text{lib}}{\Phi \langle \text{call}_k i \rangle_{\text{lib}} \rightsquigarrow \Phi[\text{stack} := [\textcolor{blue}{SF}] \mathbin{++} \Phi.stack, pc := n', sp := sp', M := M']} \\
\\
\frac{\text{is-ret-addr-loc}(\Phi, \Phi.sp) \quad \langle n \rangle = \Phi.M(\Phi.sp) \quad n' = k(n) \quad \text{csr-restored}(\Phi) \quad \Phi' = \text{pop-frame}(\Phi)}{\Phi \langle \text{ret}_k \rangle_{\text{lib}} \rightsquigarrow \Phi'[pc := n', sp := \Phi.sp - 1]} \\
\\
\frac{n' \in I \quad \langle n' \rangle = \mathcal{V}_{\Phi}(i) \quad sp' = \Phi.sp + 1 \quad M' = \Phi.M[sp' \mapsto \Phi.pc + 1] \quad \text{typechecks}(\Phi, n', sp') \quad \text{args-secure}(\Phi, sp', n) \quad \textcolor{blue}{SF} = \text{new-frame}(\Phi, n', sp')}{\Phi \langle \text{gatecall}_n i \rangle_{\text{lib}} \rightsquigarrow \Phi[\text{stack} := [\textcolor{blue}{SF}] \mathbin{++} \Phi.stack, pc := n', sp := sp', M := M']} \\
\\
\frac{M' = \Phi.M[sp' \mapsto \Phi.pc + 1] \quad \langle n' \rangle = \mathcal{V}_{\Phi}(i) \quad sp' = \Phi.sp + 1 \quad \textcolor{blue}{SF} = \text{new-frame}(\Phi, n', sp')}{\Phi \langle \text{gatecall}_n i \rangle_{\text{app}} \rightsquigarrow \Phi[\text{stack} := [\textcolor{blue}{SF}] \mathbin{++} \Phi.stack, pc := n', sp := sp', M := M']} \quad \frac{\Phi \langle \text{ret} \rangle \rightsquigarrow \Phi' \quad p' \sqsubseteq p \quad \langle n, p' \rangle = \Phi.R(r_{\text{ret}})}{\Phi \langle \text{gateret} \rangle_p \rightsquigarrow \Phi'} \\
\\
\frac{v = \mathcal{V}_{\Phi}(i) \quad sp' = \Phi.sp + 1 \quad sp' \in S_p \quad M' = \Phi.M[sp' \mapsto v] \quad \text{writeable}(\Phi, sp')}{\Phi \langle \text{push}_p i \rangle \rightsquigarrow \Phi^{++}[sp := sp', M := M']} \quad \frac{\langle n, p_i \rangle = \mathcal{V}_{\Phi}(i) \quad v = \langle _, p_{i'} \rangle = \mathcal{V}_{\Phi}(i') \quad M' = \Phi.M[n' \mapsto v] \quad \text{writeable}(\Phi, n') \quad n' = k(n) \quad p_i \sqsubseteq p \quad p_{i'} \not\sqsubseteq p \implies n' \notin H_{\text{lib}}}{\Phi \langle \text{store}_k i := i' \rangle_p \rightsquigarrow \Phi^{++}[M := M']} \\
\\
\frac{\langle n, p_i \rangle = \mathcal{V}_{\Phi}(i) \quad n' = k(n) \quad p_i \sqsubseteq p \quad v = \langle _, p_{n'} \rangle = \Phi.M(n') \quad R' = \Phi.R[r \mapsto v]}{\Phi \langle r \leftarrow \text{load}_k i \rangle_p \rightsquigarrow \Phi^{++}[R := R']} \quad \frac{\langle v, p_i \rangle = \mathcal{V}_{\Phi}(i) \quad p_i \sqsubseteq p}{\Phi \langle sp \leftarrow \text{mov } i \rangle_p \rightsquigarrow \Phi^{++}[sp := v]} \\
\\
\frac{\langle n, p_i \rangle = \mathcal{V}_{\Phi}(i) \quad n' = k(n) \quad p_i \sqsubseteq p \quad \text{in-same-func}(\Phi, \Phi.pc, n')}{\Phi \langle \text{jmp}_k i \rangle_p \rightsquigarrow \Phi[pc := n']} \quad \frac{p_r \not\sqsubseteq p' \implies p_r \sqsubseteq p \quad \langle n, p_r \rangle = \Phi.R(r) \quad R' = \Phi.R[r := \langle n, p' \rangle]}{\Phi \langle r \leftarrow \text{movlabel}_{p'} \rangle_p \rightsquigarrow \Phi^{++}[R := R']} \\
\\
\frac{\langle n, p_i \rangle = \mathcal{V}_{\Phi}(i) \quad \langle m, p_m \rangle = \Phi.M(n) \quad p_i \sqsubseteq p \quad M' = \Phi.M[n := \langle m, p' \rangle] \quad p_m \not\sqsubseteq p' \implies p_m \sqsubseteq p}{\Phi \langle \text{storelabel}_{p'} i \rangle_p \rightsquigarrow \Phi^{++}[M := M']} \quad \frac{\Phi.\Psi \langle c \rangle \rightarrow \Psi' \quad \text{in-same-func}(\Phi, \Phi.\Psi.pc, \Psi'.pc)}{\Phi \langle c \rangle \rightsquigarrow \Phi[\Psi := \Psi']}
\end{array}$$

Figure 19: Overlay Operational Semantics

$$\begin{array}{c}
\frac{[SF] \uparrow _ = \Phi.stack \quad n \in S_p \implies n \geq SF.base \wedge n \neq SF.ret-addr-loc}{\text{writeable}(\Phi, n)} \quad \frac{F = \Phi.funcs(target) \quad F.entry = target \quad sp \in S_p \quad [SF] \uparrow _ = \Phi.stack \quad sp \geq SF.ret-addr-loc + F.type}{\text{typechecks}(\Phi, target, sp)} \\
\\
\frac{[SF] \uparrow _ = \Phi.stack \quad ret-addr-loc = SF.ret-addr-loc}{\text{is-ret-addr-loc}(\Phi, ret-addr-loc)} \quad \frac{[SF] \uparrow _ = \Phi.stack \quad \forall (r, n) \in SF.csr-vals. \Phi.R(r) = n}{\text{csr-restored}(\Phi)} \quad \frac{F \in \text{cod}(\Phi.funcs) \quad n, n' \in \text{dom}(F.instrs)}{\text{in-same-func}(\Phi, n, n')} \\
\\
\frac{\forall F \in \text{cod}(\Phi.funcs). n \notin \text{dom}(F.instrs)}{\text{in-same-func}(\Phi, n, n')} \quad \frac{\forall i \in [1, n]. \Phi.M(sp - i) = \langle _, \text{lib} \rangle}{\text{args-secure}(\Phi, sp, n)}
\end{array}$$

Figure 20: Overlay Operational Semantics: Auxiliary Judgments

$$\begin{aligned}
\text{new-frame}(\Phi, target, ret-addr-loc) &\triangleq \{ \begin{array}{ll} \text{base} &= ret-addr-loc - \Phi.funcs(target).type \\ \text{ret-addr-loc} &= ret-addr-loc \\ \text{csr-vals} &= \{(r, \Phi.R(r))\}_{r \in \text{CSR}} \end{array} \} \\
\text{pop-frame}(\Phi) &\triangleq \Phi[stack := S] \quad \text{where } [SF] \uparrow S = \Phi.stack \\
\Phi^{++} &\triangleq \begin{cases} \Phi[\Psi := \Psi^{++}] & \text{in-same-func}(\Phi, \Phi.pc, \Phi.pc + 1) \\ \text{error} & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 21: Overlay Operational Semantics: Auxiliary Definitions

Lemma 9 (Overlay is a refinement). *For any $\Phi \rightsquigarrow \Phi'$, if $\Phi' \neq \text{error}$, then $\Phi.\Psi \rightarrow \Phi'.\Psi$*

Lemma 10 (Overlay is equivalent on application reduction). *For any Φ , if $\Phi.\Psi \xrightarrow{\text{app}} \Psi'$, then $\Phi \rightsquigarrow \{\Psi := \Psi', funcs := \Phi.funcs, stack := \Phi.stack\}$.*

Theorem 4 (Overlay Integrity Soundness). *If $\Phi_0 \in \text{Program}$, $\Phi_0 \rightsquigarrow^n \Phi_1$, $\Phi_1(_)_{\text{app}}$, 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) $\mathcal{CSR}(\pi, \Phi_1.\Psi, \Phi_2.\Psi)$
- 2) $\mathcal{RA}(\pi, \Phi_1.\Psi, \Phi_2.\Psi)$

Proof. By induction over the definition of a well-bracketed step and nested induction over the logical call stack. The last step follows by the fact that $\Phi_2 \neq \text{error}$, and therefore the restoration checks in the overlay monitor passed. \square

Theorem 5 (Overlay Confidentiality Soundness). *If $\Phi_0 \in \text{Program}$, $\Phi_1(_)_{\text{lib}}$, $\Phi_3(_)_{\text{app}}$, $\Phi_0.\Psi \rightarrow^* \Phi_1.\Psi \xrightarrow{\text{lib}}^n \Phi_2.\Psi \rightarrow \Phi_3.\Psi$, $\Phi_1 \rightsquigarrow^{n+1} \Phi_3$, and $\Phi_1 =_{\text{lib}} \Phi'_1$, then $\Phi'_1.\Psi \xrightarrow{\text{lib}}^n \Phi'_2.\Psi \rightarrow \Phi'_3.\Psi$, $\Phi'_1 \rightsquigarrow^{n+1} \Phi'_3$, $\Phi'_3(_)_{\text{app}}$, $\Phi_3.pc = \Phi'_3.pc$, and*

- 1) $\Phi_2(\text{gatecall}_{n'} i)$, $\Phi'_2(\text{gatecall}_{n'} i)$, and $\Phi_3 =_{\text{call } n'} \Phi'_3$ or
- 2) $\Phi_2(\text{gateret})$, $\Phi'_2(\text{gateret})$, and $\Phi_3 =_{\text{ret}} \Phi'_3$,

Proof. Proof is standard for an IFC enforcement system. \square

APPENDIX D WEBASSEMBLY

$$\begin{array}{ccc}
\frac{\Psi(\text{call } i) \rightarrow \Psi'}{\Psi(\text{gatecall}_n i)_{\text{app}} \rightarrow \Psi'[p := \text{lib}]} & \frac{\Psi(\text{call}_I i) \rightarrow \Psi'}{\Psi(\text{gatecall}_n i)_{\text{lib}} \rightarrow \Psi'[p := \text{app}]} & \frac{\Psi(\text{ret}) \rightarrow \Psi' \quad p' = \text{lib} \Leftrightarrow p = \text{app}}{\Psi(\text{gateret})_p \rightarrow \Psi'[p := \text{app}]}
\end{array}$$

Figure 22: WebAssembly Trampoline and Springboard

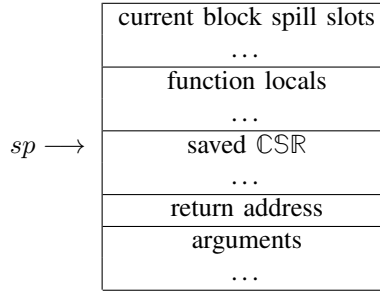


Figure 23: WebAssembly Stack Frame

$$\begin{aligned}
 \text{WebAssemblyFunction } WF &::= \{ \begin{array}{ll} |args| &: \mathbb{N} \\ |locals| &: \mathbb{N} \\ entry &: Block \\ exits &: \wp(Block) \\ blocks &: \wp(Block) \end{array} \} \\
 Block \ B &::= \{ \begin{array}{ll} start &: \mathbb{N} \\ end &: \mathbb{N} \\ |slots| &: \mathbb{N} \\ inputs &: \wp(\mathbb{N} + Reg) \\ indirects &: \wp(IB) \end{array} \} \\
 Indirect\ Block \ IB &::= \{ \begin{array}{ll} start &: \mathbb{N} \\ end &: \mathbb{N} \\ parent &: Block \end{array} \}
 \end{aligned}$$

Figure 24: WebAssembly Structure

A compiled WebAssembly library is divided into disjoint functions F which have a fixed number of arguments and locals. Each function is composed of a set of disjoint blocks B where the block's code is $C([B.start, B.end])$, with a distinguished entry block and set of exit blocks. There are a fixed number of stack slots associated to each block along with a subset of the slots or registers assigned as the inputs to the block. Each non-exit block is terminated ($C(B.end)$) by a branch or jump. The exit blocks in a function are uniformly terminated by a $\text{ret}_{C_{13b}}$ or gateret . To simplify things, we will assume that WebAssembly functions cannot call a function whose exit blocks terminate in gateret and the application code always calls such a function. Branches and jumps cannot appear anywhere except at the end of a block.

Call instructions only appear in a block within one of two sequences of instructions. The first correspond to direct calls in WebAssembly:

$$\begin{aligned}
 &sp \leftarrow sp + (F.|locals| + B.|slots|); \\
 &\text{push } arg_1; \dots; \text{push } arg_{F'.|args|}; \\
 &\text{call } F'.entry.start; \\
 &sp \leftarrow sp - (F.|locals| + B.|slots| + F'.|args|)
 \end{aligned}$$

The second correspond to indirect calls each of which have an associated expected type of the callee:

$$\begin{aligned}
 &\text{typecheck}(r, |expected-args|); \\
 &sp \leftarrow sp + (F.|locals| + B.|slots|); \\
 &\text{push } arg_1; \dots; \text{push } arg_{|expected-args|}; \\
 &\text{call}_{\mathcal{T}} r; \\
 &sp \leftarrow sp - (F.|locals| + B.|slots| + |expected-args|)
 \end{aligned}$$

\mathcal{T} is the set of entry points to functions corresponding to the WebAssembly table used by this indirect call, and typecheck is implementation specific code that checks that the function at the address in register r expects $|expected-args|$.

In both cases the call instruction can be replaced with a gated call instruction:

```

 $sp \leftarrow sp + (F.|locals| + B.|slots|);$ 
push  $arg_1; \dots; \text{push } arg_n;$ 
gatecall $_n v;$ 
 $sp \leftarrow sp - (F.|locals| + B.|slots| + n)$ 

```

```

typecheck( $r, n$ );
 $sp \leftarrow sp + (F.|locals| + B.|slots|);$ 
push  $arg_1; \dots; \text{push } arg_n;$ 
gatecall $_n r;$ 
 $sp \leftarrow sp - (F.|locals| + B.|slots| + n)$ 

```

v is an address in I that expects n arguments.

There are three kinds of memory accesses allowed within a block. Firstly, heap accesses which are guarded to be in the sandboxed heap: $r \leftarrow \text{load}_{H_{1b}} i$ and $\text{store}_{H_{1b}} i := i'$. Secondly, argument, local, or variable accesses which are addressed by constant offsets from the stack pointer: $r \leftarrow \text{load } sp + x$ or $\text{store } sp + x := i$ where $x \in [-(\mathbb{C}\mathbb{S}\mathbb{R}|_F + F.|args|), -\mathbb{C}\mathbb{S}\mathbb{R}|_F] \cup [1, F.|locals| + B.|slots|]$. Lastly, initializing a block's inputs before jumping. These immediately precede a constant branch or jump instruction to some $B'.start$: $\text{store } sp + x := i$ where $x \in F.|locals| + B'.inputs$. Any reads from stack slots or registers are preceded by writes to that stack slot or register within the block or the stack slot or register is in the block's input set $B.inputs$.

WebAssembly's indirect branches are compiled to an indirect jump to an *Indirect Block (IB)*. These blocks are terminated by a constant jump instruction to a block B and consist entirely of initializing the inputs to B . An indirect block IB may read variables according to the rules of its parent block ($IB.parent$). As such an indirect jump at the end of a block B is guarded:

$\text{jmp}_{B.indirects.start} r$.

Lastly, a WebAssembly function F 's entry block begins by pushing the subset of the callee-save registers in $F.blocks$ (written $\mathbb{C}\mathbb{S}\mathbb{R}|_F$) and each exit block pops these back into their corresponding register.

A. Logical Relation

$$L ::= \left\{ \begin{array}{ll} \text{interface} & : \mathbb{N} \rightarrow \mathbb{N} \\ \text{library} & : \mathbb{N} \rightarrow \text{WasmFunction} \\ \text{code} & : \text{Code} \end{array} \right\}$$

$$\text{World} \triangleq \mathbb{N} \times (\mathbb{N} \rightarrow \mathbb{N}) \times (\mathbb{N} \rightarrow \mathbb{N})$$

$$\begin{aligned} \triangleright & : \text{World} \rightarrow \text{World} \\ \triangleright(0, \overline{f_i}, \overline{f_l}) & \triangleq (0, \overline{f_i}, \overline{f_l}) \\ \triangleright(n, \overline{f_i}, \overline{f_l}) & \triangleq (n-1, \overline{f_i}, \overline{f_l}) \end{aligned}$$

$$\begin{aligned} (C, \overline{F_2}, \overline{F_3}) :_W & \triangleq \text{for } i \in \{1, 2\} : \\ & \overline{F_i}.entry = \text{dom}(\pi_i(W)) \\ & \forall \overline{F_i} \in \overline{F_i}. \overline{F_i}.type = \pi_i(W)(\overline{F_i}.entry) \\ & \forall \overline{F_i} \in \overline{F_i}. (\triangleright W, \overline{F_i}, C|_{\overline{F_i}.instrs}) \in \mathcal{F} \end{aligned}$$

$$\begin{aligned}
\mathcal{F} &\triangleq \left\{ (W, \mathbf{F}, c) \left| \begin{array}{l} \forall \rho \in [\text{Frame}], \text{ret-addr}, A \in [\mathbb{N}], sp, R, M, C, \overline{F_i}, \overline{F_l}. \\ \text{let } \rho' = \rho \uparrow \{ \text{base} := sp - |A|, \text{ret-addr-loc} := sp, \text{csr-vals} := R(\text{CSR}) \} \\ |A| = \mathbf{F}.type \\ sp > \text{top}(\rho). \text{ret-addr-loc} + |A| \\ [sp \mapsto \text{ret-addr}, (sp - |A| + i \mapsto A_i)_{i \in [0, |A|)}] \leq M \\ (C, \overline{F_i}, \overline{F_l}) :_W \\ c \leq C \\ \text{dom}(c) = \mathbf{F}.instrs \\ \forall \ell \in \text{dom}(M) \cap M_{\text{lib}}. \langle n, \text{lib} \rangle = M(\ell) \\ \text{let } \Psi = \{ pc := \mathbf{F}.entry, sp := sp, R := R, M := M, C := C \} \\ \text{let } \Phi = \{ \Psi := \Psi, \text{stack} := \rho', \text{funcs} := [\mathbf{F}'.entry \mapsto \mathbf{F}']_{\mathbf{F}' \in \overline{F_i} \uplus \overline{F_l}} \} \\ \implies \\ \forall n' \leq W.n. \Phi \xrightarrow{\rho'}^{n'} \Phi' \implies \Phi' \neq \text{error} \\ \text{or } \exists n' \leq W.n. \Phi \xrightarrow{\rho'}^{n'-1} \Phi' \rightsquigarrow \Phi'' \\ \text{where } (\Phi'(\text{ret})) \vee \Phi'(\text{gateret})) \wedge \Phi'.\text{stack} = \rho' \wedge \Phi' \neq \text{error} \end{array} \right. \right\} \\
\mathcal{L} &\triangleq \left\{ (n, L) \left| \begin{array}{l} \forall i \in \text{dom}(L.\text{library}). \\ \text{let } WF = L.\text{library}(i) \\ \text{let } W = (n, L.\text{interface}, \lambda i \rightarrow L.\text{library}(i).|\text{args}|) \\ \text{let } instrs = \biguplus_{B \in WF.\text{blocks}} [B.\text{start}, B.\text{end}] \\ \text{let } \mathbf{F} = \{ instrs := instrs, entry := WF.\text{entry}.\text{start}, type := WF.|\text{args}| \} \\ (W, \mathbf{F}, L.\text{code}|_{instrs}) \in \mathcal{F} \end{array} \right. \right\}
\end{aligned}$$

Figure 25: Function and Library Relations

Lemma 11 (FTLR for functions). *Let $W \in \text{World}$ and c be the code for a compiled WebAssembly function WF such that WF expects application functions in the interface with locations and types $\pi_2(W)$ and in the library with locations and types $\pi_3(W)$. Further let $instrs = \biguplus_{B \in WF.\text{blocks}} [B.\text{start}, B.\text{end}]$ and $\mathbf{F} = \{ instrs := instrs, entry := WF.\text{entry}.\text{start}, type := WF.|\text{args}| \}$. Then $(W, \mathbf{F}, c) \in \mathcal{F}$.*

Proof. We first unroll the assumptions of $(W, \mathbf{F}, c) \in \mathcal{F}$ reusing the variable names defined there. We will maintain that any steps do not step to **error** so WOLOG we will continually assume $n' \leq W.n$ such that n' greater than the number of steps we have taken, otherwise the case $\Phi \xrightarrow{\rho'}^{n'} \Phi' \implies \Phi' \neq \text{error}$ holds.

By the structure of a compiled WebAssembly function and assumption we have that the stack and stack pointer represent $WF.|\text{args}|$ arguments. The structure of an entry block means that we begin by pushing $\text{CSR} \cap R_c$ where R_c is all registers mentioned in c . After this we generalize over which block in $WF.\text{blocks}$ we are in, adding the antecedent that we do not touch the return address or callee-save registers, and then continue with our proof. The structure of a compiled WebAssembly block then lets us proceed until we reach one of 1) a function call to a library function WF' such that $WF'.\text{entry}.\text{start} \in \overline{F_l}.\text{entry}$, 2) an application function \mathbf{F}' such that $\mathbf{F}'.\text{entry} \in \overline{F_i}.\text{entry}$, 3) or the end of the block.

- 1) We have by assumption that we have pushed $WF'.|\text{args}| = \pi_3(W)(WF'.\text{entry}.\text{start})$ arguments or failed a dynamic type check and terminated (thus stepping to a terminal state that is not an **error**). We thus set $\rho_2 = \rho'$ and see that we have constructed $\rho'_2 = \rho_2 \uparrow \{ \text{base} := sp - WF'.|\text{args}|, \text{ret-addr-loc} := sp, \text{csr-vals} := R(\text{CSR}) \}$. We further set $\text{ret-addr} = pc + 1$, $A = WF'.|\text{args}|$, $sp = sp$, $R = R$, $M = M$, $C = C$, $\overline{F_i} = \overline{F_i}$, and $\overline{F_l} = \overline{F_l}$. By the structure of compiled WebAssembly we have that all of the remaining checks in \mathcal{F} pass and that the instantiated Φ is equal to our current state. We therefore instantiate $(\triangleright \overline{F_l}, C|_{\overline{F_l}.\text{instrs}}) \in \mathcal{F}$. If this uses the remaining steps then we are done. Otherwise we get that we return to $pc + 1$ with all values restored and no new app values written to the library memory, and our walk through the block may continue.
- 2) Identical to the case for (1).
- 3) The end of a block is followed by a direct jump to another block B' , an indirect block IB , or we are at an exit block. In the case of another block B' we have by the structure of compiled WebAssembly code that we have instantiated $B'.\text{inputs}$. We thus jump to the block and follow the same proof structure as detailed here. The same is true of an intermediate block IB except with the extra steps of setting up the inputs jumping to another block B'' . Lastly if we have reached the end of an exit block then we have not touched the pushed return address or callee-save registers and the stack pointer is in the expected location. We thus execute **ret** or **gateret** and pass the overlay monitor checks.

□

Lemma 12 (FTLR for libraries). *For any number of steps $n \in \mathbb{N}$ and compiled WebAssembly library L , $(n, L) \in \mathcal{L}$.*

Proof. By unrolling the definition of \mathcal{L} and Lemma 11 □

Theorem 6 (Adequacy of \mathcal{L}). *For any number of steps $n \in \mathbb{N}$, library L such that $(n, L) \in \mathcal{L}$, program $\Phi_0 \in \text{Program}$ using L , and $n' \leq n$, if $\Phi_0 \rightsquigarrow^{n'} \Phi'$ then $\Phi' \neq \text{error}$.*

Proof. Straightforward: by assumption for steps in the application, and by assumption about application code properly calling the library code and the unrolling of \mathcal{L} and \mathcal{F} . □

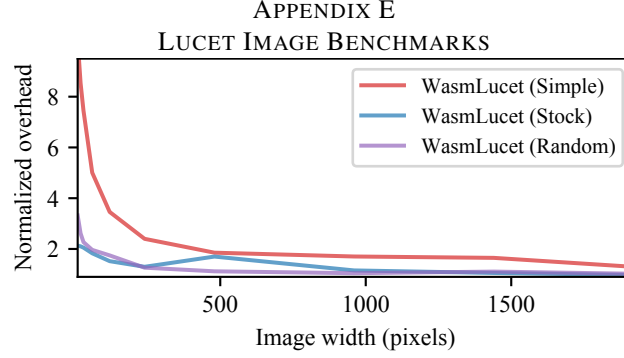


Figure 26: Performance of the WasmLucet heavyweight transitions included in the Lucet runtime on the image benchmarks in Section VI. Performance when rendering (a) a simple image with one color, (b) a stock image and (c) a complex image with random pixels. The performance is the overhead compared to WasmZero.