Доверя́й, но проверя́й: SFI safety for native-compiled Wasm

Evan Johnson† David Thien† Yousef Alhessi† Shravan Narayan†
Fraser Brown* Sorin Lerner† Tyler McMullen◇ Stefan Savage† Deian Stefan†
†UC San Diego *Stanford ◇Fastly Labs

***Abstract*—WebAssembly (Wasm) is a platform-independent bytecode that offers both good performance and runtime isolation. To implement isolation, the compiler inserts safety checks when it compiles Wasm to native machine code. While this approach is cheap, it also requires trust in the compiler's correctness—trust that the compiler has inserted each necessary check, correctly formed, in each proper place. Unfortunately, subtle bugs in the Wasm compiler can break—and *have broken*—isolation guarantees. To address this problem, we propose verifying memory isolation of Wasm binaries post-compilation. We implement this approach in VeriWasm, a static offline verifier for native x86-64 binaries compiled from Wasm; we prove the verifier's soundness, and find that it can detect bugs with no false positives. Finally, we describe our deployment of VeriWasm at Fastly.**

I. Introduction

WebAssembly (Wasm) is a modern, platform-independent bytecode that was originally designed to be embedded in the browser, and was therefore designed with isolation in mind. Ironically, the fact that Wasm's design was tied to the browser also unshackled it *from* the browser: different "embedding environments" are using Wasm as a general-purpose software-based fault isolation (SFI) mechanism.

For example, the Fastly and Cloudflare content-delivery networks (CDNs) use Wasm to isolate different tenants on their edge clouds [77], [55]. They run many clients' Wasm code within a single process, using Wasm-based SFI to protect clients from each other—and to protect the embedding host process from clients. Any break in Wasm's isolation guarantees could allow a malicious client to steal or corrupt data that belongs to another client (or the host). Similarly, Mozilla uses Wasm to sandbox third party C libraries in the Firefox renderer [49], [50]. The Wasm compiler—in this case, the Bytecode Alliance Lucet compiler [55]—essentially serves to instrument the library with inline SFI checks that restrict the library control and data flows to its own sandbox. But if this Wasm-based SFI fails, everyday web users are vulnerable to remote attackers, who could exploit such "sandbox escape" bugs to do anything from steal data to fully compromise machines.

In general, all systems that use Wasm for isolation implicitly trust the Wasm compiler with their users' safety—the compiler is solely responsible for enforcing isolation by inserting checks into the native code it generates. However, compilers are complex software artifacts and code optimization passes are especially notorious for introducing unintended consequences [82], [57], [71], [83], [36], [69]. Wasm compilers perform optimizations *after* inserting memory safety checks,[1] so any bugs in optimization could allow a carefully generated Wasm program to bypass Wasm's safety checks. These bugs actually happen: a bug in the Lucet loop invariant code motion pass broke SFI-safety, allowing maliciously-crafted Wasm code to escape the sandbox to, say, read arbitrary memory [26]. Though there's an active research community focused on building verified compilers (e.g., [40], [18])—even a verified Wasm compiler [13]!—it seems likely that defects in industrial compilers will persist for some time.

Given this practical reality, we propose an alternative method for ensuring the safety of Wasm code: *verifying* the isolation of native-compiled Wasm modules using a verifier that operates directly on the generated machine code. This approach has a number of benefits. First, by working with native binaries, the verifier engages directly with the "final form" of a module and does not need to consider any further transformations to the instruction stream. Second, the verifier is much simpler than a compiler or run-time system, which not only shrinks the trusted computing base, but also makes the verifier easier to prove correct using formal methods. Finally, the memory-safety verifier runs once per Wasm module, and thus imposes no run-time overhead.

Our work makes the following contributions:

- A framework that describes sufficient conditions for reasoning about the memory-safety properties of native-compiled Wasm modules at the x86-64 instruction level.
- The design and implementation of VeriWasm, a static offline verifier that uses this framework to check the memory safety of x86-64 binaries produced by the Lucet Wasm compiler.
- A formal proof that our verifier is *sound*, i.e., it only labels programs safe if they are indeed safe. We mechanically verify our proof using the Coq theorem prover.
- An empirical assessment of VeriWasm against both benign and faulty native-compiled Wasm modules, including two Wasm-sandboxed libraries used in Firefox and 103 client binaries deployed on Fastly's production edge cloud.

Our code—both VeriWasm and our Coq model and proofs—and data are available under an open source licence [29].

II. A brief intro to Wasm and its ancestors

The promise of safe, portable binary code is one that has driven well over 25 years of research. We briefly summarize

[1] Note that this choice is on purpose, as subsequent optimization passes may allow redundant or unnecessary checks to be removed.

Network and Distributed Systems Security (NDSS) Symposium 2020
23-26 February 2020, San Diego, CA, USA
ISBN 1-891562-61-4
https://dx.doi.org/10.14722/ndss.2020.23xxx
www.ndss-symposium.org

this history to place Wasm in context, give an overview of Wasm's design, and finally explain how VeriWasm can help.

A. Ancestral systems

The idea of creating portable execution formats, independent of high-level language or machine architecture, is at least sixty years old—going back to Conway's UNCOL effort in the late 1950s [19]. But, in spite of a range of interesting systems (e.g., notably UCSD's p-System [16]), the idea did not take off until the 1990s, which saw the introduction of the Open Software Foundation's Architecture Neutral Distribution Format [44], Sun's Java Virtual Machine [42], and Microsoft's Common Language Runtime [46].

Of these, Java was one of the first systems to address the need for safety. Java included both a load-time verifier and run-time checks designed to ensure that maliciously constructed Java bytecode could not bypass the language's type-and memory-safety guarantees. This proved challenging in practice, in part because Java's complex type system in turn demanded a complex verifier—and, unfortunately, verifier bugs led to a broad range of "sandbox escapes" [6].

In 1994, Wahbe et al. introduced the notion of *Software-based Fault Isolation* (SFI) [80], a family of binary instrumentation techniques for ensuring coarse-grained memory safety. While Wahbe and colleagues integrated their original SFI concept into the Omniware mobile code format [4], most subsequent efforts focused on developing SFI in the context of existing instruction set architectures (e.g., x86, x86-64) [66], [23], [45], [84], [62]. Several of these, including the original SFI scheme, were designed with load-time verification in mind, i.e., they properly instrumented binaries to not only satisfy all memory-safety invariants, but make it easy to verify these invariants. Google's NativeClient (NaCl) is perhaps the most popular of these systems—in part due to its integration in the Chrome browser—and its PNaCl variant included a portable intermediate binary format [22], like the original Omniware.

WebAssembly, first announced by the W3C in 2015, is an effort to mainstream much of this past work and produce a standard high-performance machine-independent bytecode that is also safe [25].

B. Wasm: Fast, modern SFI

Wasm was explicitly designed to be easy to sandbox—after all, browsers need to parse, validate, and compile Wasm within a page load. Indeed, many of Wasm's design choices—from its static type system to its structured control flow and memory hierarchy—are in service of sandboxing: they make it easy for a compiler to generate native code that is SFI-safe, i.e., code whose data- and control-flow are isolated to the sandbox. These design choices are also the reason organizations like Fastly and Mozilla are starting to use Wasm to sandbox potentially untrusted code: by compiling to Wasm, they get SFI for free.

Control-flow safety. Wasm compilers must ensure that the control flow of the generated native code is safely restricted to the sandbox. Two Wasm design features simplify this task. First, Wasm exposes structured control instructions (e.g., `if` blocks and `loop`s) that preserve Wasm's type safety; Wasm does not expose unstructured control flow instructions like `goto` that would make this task harder. Second, the bytecode only allows indirect control transfers via static look-up tables: `call_indirect`, for example, is used to call functions registered in the module indirect-function table; `br_table` is used to jump to local branch-table blocks. Both of these instructions perform dynamic checks to ensure control is transferred to a valid function or block of the correct type, respectively. They also make it easy to ensure the module's indirect control flow is safe when compiled to native code. For example, when compiling `call_indirect idx` to x86-64, the Lucet compiler simply ensures that the index (`idx`) corresponds to a registered function, checks the type of the function at this index, and then calls the function.

Memory isolation. Wasm compilers must also ensure memory isolation. Again, deliberate language design choices simplify this task: Wasm exposes three distinct isolated memory regions—the stack, global variables, and a linear memory region—which must be accessed with different type-safe instructions. This makes it easy for a compiler to ensure that memory accesses are safe when compiling to native code. Instructions that access stack variables (`local.get/set idx`) or global variables (`global.get/set idx`) can simply be compiled to native code that access memory at constant offsets (`idx`) from the stack pointer and global variable memory base, respectively. Similarly, linear memory accesses (`load/store offset`) can be compiled into native loads and stores that access memory at `offset` off the linear memory base, after appropriately ensuring the `offset` is within the linear memory bound.

C. Trust but verify

While Wasm makes it easy for compilers to enforce SFI, we still have to *trust* compilers to do so correctly. Correct compilation is easier said than done. Modern optimizing compilers are complex, and a single bug in an optimization pass could result in a sandbox escape. On a Wasm edge-cloud, this could, for example, allow an attacker to steal or corrupt data sensitive to other clients or the cloud provider (e.g., SSL keys). In the browser, where Wasm is used to sandbox libraries, it could allow an attacker to compromise the renderer to, again, steal or corrupt sensitive data.

We need a way to *verify* that Wasm compilers preserve SFI even across optimization passes. One way to do this is to prove that the compiler is correct (e.g., like the Certified CompCert C compiler [40]). Alas, verifying an industrial optimizing compilers is notoriously hard (e.g., it took CompCert 100,000 lines of Coq and 6 person-years for the proof alone [41]). Luckily, we don't need to prove a compiler correct to ensure that the programs it produces are safe. Instead, like NaCl, we can verify that the compiler inserts the necessary checks to enforce SFI safety in each binary it produces. In this paper, we describe such a verifier for the Lucet compiler.

III. VeriWasm overview

VeriWasm is a static SFI verifier for native-compiled Wasm. VeriWasm takes as input a possibly buggy or malicious native-compiled Wasm module, and uses a sound static analysis to determine if the module is safe. In this section, we give a brief overview of VeriWasm's design and the four local safety properties it verifies. In Section IV we describe the static

analysis passes that verify these properties, and in Section V, we formally verify the soundness of the verifier.

VeriWasm design. VeriWasm verifies binaries produced by the Lucet WebAssembly compiler. Fig. 1 gives an overview of VeriWasm's different stages. The tool first recursively disassembles the native-compiled Wasm binary and produces a control-flow graph for every function exposed in the symbol table. As a part of this process, VeriWasm also resolves all indirect jumps in the control-flow graph (see Section IV-D1), and ensures that all direct and indirect calls target functions present in the symbol table. Then, VeriWasm checks the disassembled code against a list of safe native instructions—the instructions the Lucet compiler emits—and rejects binaries with potentially unsafe instructions (like `int` and `syscall`). Finally, VeriWasm analyzes each function to verify local safety properties; if all functions are safe, VeriWasm then declares the module safe.

VeriWasm's local safety properties. To verify if a compiled function is safe—that its control- and data- flow is isolated to the Wasm module—VeriWasm verifies *four local safety properties*:

- *Linear memory isolation*: All linear memory accesses must fall within the linear memory bounds (or surrounding guard pages). This ensures that linear memory operations cannot span beyond the sandbox boundary.
- *Stack isolation and integrity*: All stack accesses must fall within the stack region (or surrounding guard pages) and all stack writes must be restricted to local variables in the current stack frame. This ensures that stack accesses cannot extend past the sandbox and prevents a function from writing past the local variables stored on the stack (e.g., return addresses, arguments, and other frames).
- *Global variable isolation*: All global variable accesses must fall within the global variable memory region. This ensures global variable accesses cannot extend past the sandbox.
- *Control flow safety*: All indirect jumps must target valid code blocks, all indirect calls must target the start of valid functions, and all returns must return to the calling function. This coarse-grained control flow integrity (CFI) ensures that only VeriWasm-verified code runs.

These safety properties are sufficient to prove that native-compiled Wasm binaries are safe (§V). But these safety properties are also less restrictive than Wasm itself. For example, Wasm's CFI restricts indirect calls to target functions of the right type; our control flow safety property, on the other hand, only requires indirect calls to target valid functions (see §IV-D2 for further discussion). Wasm similarly restricts functions from reading beyond their stack frame, while our stack isolation property allows functions that read the whole stack. This difference is important: it simplifies the analysis (e.g., by not requiring it to perform type inference to enforce Wasm's finer-grain, type-based CFI). We describe our analysis—which exploits Lucet's implementation choices for additional simplicity—next.

IV. VeriWasm's analysis

VeriWasm uses abstract interpretation to verify the isolation of Wasm modules compiled with Lucet. Abstract interpretation is a static analysis technique that infers information about a program by overapproximating its behavior. It executes the code similar to a standard interpreter, but describes a program's variables as abstract values representing a set of concrete values relevant to a safety property of the variable. For example, VeriWasm's linear memory safety analysis tracks whether variables in registers and on the stack are less than 2^{32} which represent valid offsets into the linear memory.

Each of VeriWasm's safety analyses have different abstract semantics; however, they all have some characteristics in common. The first is that they only track variables in registers and on the stack, but not in the linear memory. VeriWasm can validate modules without tracking variables across the linear memory because Lucet's optimizations also do not track variables across the linear memory. The second characteristic is that all of VeriWasm's analysis passes analyze each function independently. When VeriWasm encounters a call instruction, it skips over the call, but acts as if the callee modified every register—they must be checked again before the code uses them in an SFI-sensitive operation. This ensures that even if the callee modifies callee-saved registers, the caller function is still safe. The caller's stack frame and stack pointer are preserved as a consequence of stack frame integrity (§IV-B). The third characteristic is that all of VeriWasm's analyses are sound: all functions VeriWasm labels as safe are indeed safe.

Unlike previous SFI systems (e.g. [80], [84], [45]), Wasm was not designed with verification in mind—and this necessarily makes our analysis more complex. The NaCl compiler, for example, compiles code to *bundles* [84], [45][2], and ensures that all safety checks are local to a bundle and cannot be to moved (outside the bundle) or optimized away. This allows the verifier to perform a single pass analysis, at the bundle-level, to ensure both memory and control flow safety [84], [47]. In contrast, Wasm compilers perform optimizations *after* inserting SFI safety checks—and thus checks can be moved (e.g., checks can be lifted outside of a loop as part of a loop invariant code motion, as mentioned in Section I) or, when redundant, removed. This is crucial for performance, but it unfortunately means that the verifier must account for such optimizations when checking SFI safety. In the rest of this section, we describe VeriWasm's analysis passes, ending with a description of how VeriWasm validates safety in the presence of compiler optimizations.

A. Linear memory isolation

VeriWasm verifies that the module's linear memory is isolated—that all linear memory accesses fall within the module's linear memory region. Lucet gives each module a contiguous 8GB region above a linear memory base; the region is composed of 4GB of usable memory followed by a 4GB guard page. Lucet compiles all linear memory accesses to x86-64 instructions with an effective address of the form `LinearMemBase + Offset1 + Offset2`, where `Offset1` and `Offset2` are 64-bit values that should be less than 2^{32}. Addresses of this form should be at most `LinearMemBase +`

[2] A bundle is a 32-byte aligned group of instructions.

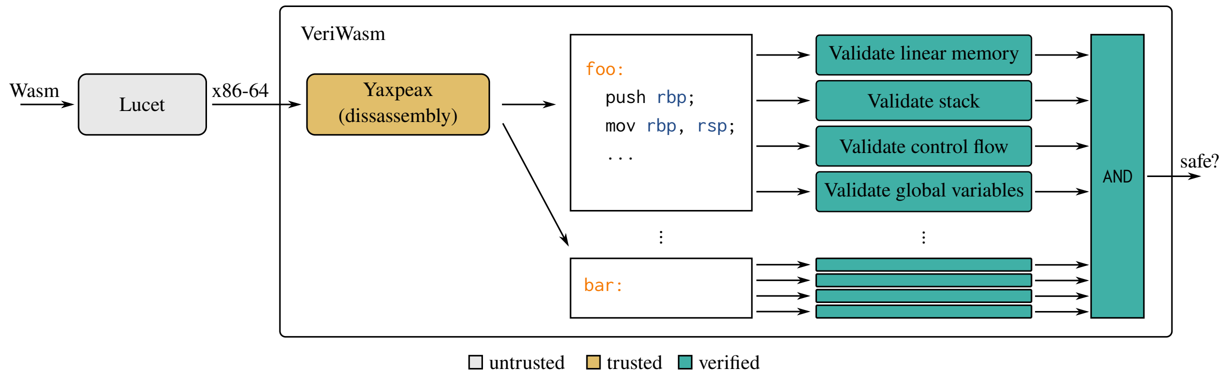

Figure 1. VeriWasm takes as input a malicious or buggy natively-compiled Wasm module and uses a trusted disassembler (Yaxpeax [3]) to create a CFG for each function in the module. VeriWasm then runs its verified analysis passes on each function's CFG to determine if the binary preserves SFI-safety.

Feature	Safety property	Description
Linear memory	Linear memory isolation	All linear memory reads and writes fall within the 4GB linear memory space (or surrounding guard pages).
Stack	Stack isolation Stack-frame integrity	Stack reads fall within the stack region (or surrounding guard pages). Stack writes are to local variables in the current stack frame.
Global variables	Global variable isolation	Global variable accesses fall within the global variable memory region.
Control flow	Jump target validity Call target validity Return target validity	All indirect jumps target valid code blocks. All indirect calls target valid functions. Functions return to their respective call sites.

Figure 2. The safety properties VeriWasm verifies to prove SFI-safety. For clarity the stack and control flow safety properties are broken down into sub-properties.

8GB, which will either be in the linear memory or the following guard page. VeriWasm ensures that all linear memory accesses are constrained to the linear memory region by verifying that: (1) `LinearMemBase` points to the linear memory base and (2) that `Offset1` and `Offset2` are less than 2^{32}.

Tracking linear memory safety. VeriWasm verifies linear memory isolation by tracking which *variables*—registers or stack slots—have a concrete value less than 2^{32} (`Bounded`), and which variables point to the start of the linear memory (`LinearMemBase`). All variables start with an abstract value of `Unknown`, except the linear memory base register (`rdi`), which starts with the value `LinearMemBase`. Other variables only become `LinearMemBase` when they're assigned from a variable with abstract value `LinearMemBase`. Variables become `Bounded` when they are truncated or assigned a 32-bit value. For example, after `mov eax, eax`, register `rax` is now `Bounded`, and after `mov rbx, 0x1337`, `rbx` is now `Bounded`.

Verifying linear memory isolation. VeriWasm applies two checks to the results of this analysis to prove that linear memory is properly isolated. VeriWasm first checks that, for all linear memory accesses, one argument is `LinearMemBase`, and the other arguments are `Bounded`. VeriWasm also verifies that at each function call, `rdi` is `LinearMemBase`. This is required by Lucet's calling convention, and violating that convention could break linear memory isolation.

The following code shows how VeriWasm verifies a function, `foo`, that reads an element in linear memory and then calls a function `bar`:

```
1  foo:
2  ; ASSUME: rdi is LinearMemBase
3  ; TRACK: rax, rbx, ... are Unknown
4    ...
5    mov eax, eax;
6    ; TRACK: rax Bounded
7    mov rsi, [rdi + rax + 0x48];
8    ; ASSERT: rdi is LinearMemBase
9    ; ASSERT: rax and 0x48 are Bounded
10   ...
11   call bar;
12   ; ASSERT: rdi is LinearMemBase
13   ...
```

The read on line 7 is safe since `rdi` is `LinearMemBase` and `rax` and `0x48` are `Bounded` (since `rax` has been truncated to 32 bits on line 5). This safety check would fail if `rdi` had been altered to point to something other than `LinearMemBase`, or if `rax` had any possibility of being greater than or equal to 2^{32}. It would also fail if `rdi` were not `LinearMemBase` in the call to bar (line 11).

B. *Stack isolation and stack-frame integrity*

VeriWasm verifies that native-compiled code cannot break isolation by misusing the stack. We do this by verifying:

- *Stack isolation*: stack reads and writes fall within the module's stack region (or surrounding guard pages, described below).
- *Stack-frame integrity*: stack writes are additionally bounded by the base of the current stack frame, i.e., stack writes cannot clobber return addresses, spilled arguments, or other function stack frames.

Before we describe how VeriWasm verifies these properties, we describe how Lucet compiles stack operations, and describe the layout of the stack region.

Native-Wasm stack instructions. Lucet compiles Wasm stack accesses into three kinds of native operations:

`rsp = rsp ± c`	stack adjustments
`x = mem[rsp ± c]`	stack loads
`mem[rsp ± c] = x`	stack stores

In this syntax, `c` is a constant: Lucet adjusts the stack by a constant amount, stores variables at a constant offset from the stack pointer, and loads variables from a constant offset from the stack pointer. VeriWasm takes advantage of the constant offsets to statically infer what part of the stack region any particular read or write falls within.

Native-Wasm stack layout. Lucet allocates a contiguous region—typically 128K—for the module stack region. Both ends of the stack region, as shown in Fig. 3, are guarded. Lucet uses a 4K guard at the end the stack region, and an 8K guard below the start of the stack region. The 4K guard region prevents most functions—all functions with fewer than 4K local variables—from growing the program stack past the stack region: at worst, accessing a local variable will land in the 4K region and trap. Some functions use more than 4K of local variables, though. For these, Lucet adds a dynamic check `probestack(k)` in the function prologue before growing the stack by `k`. The probestack ensures that growing the stack is safe, i.e., within the 128K stack region, and traps otherwise. The 8K guard at the stack region base just fits the maximum number of spilled arguments (8,000 bytes) and any control data; we require this guard region to simplify our binary analysis.[3]

Our stack safety verification is based on the key observation that a function will (1) read *at most* 8K above (towards bottom of stack region) the stack frame pointer—the function's spilled arguments; and (2) read and write *at most* 4K (or `k` if the function prologue has a `probestack(k)`) below (towards the top of the stack region) the stack frame pointer—the function's local variables. Since stack operations are in terms of the stack pointer, though, our analysis must accordingly track the difference between the stack pointer and frame pointer to verify if any particular read or write is safe.

Tracking the stack growth. VeriWasm tracks the stack growth at each point in a function. The growth, `StackGrowth`, is the difference between the stack frame pointer (the stack pointer before the function start executing) and the stack pointer after each instruction, i.e., $\texttt{StackGrowth} = \texttt{rsp}_{\texttt{current}} - \texttt{rsp}_{\texttt{start}}$. At the start of each function, `StackGrowth` is zero. Then, whenever an instruction modifies the stack, VeriWasm accordingly adjusts `StackGrowth`. For example, pushing a value to the stack decreases `StackGrowth` by eight, popping does the converse. At each merge point (e.g., the block after an `if-else`), VeriWasm checks that `StackGrowth` is the same for all incoming paths; if not, VeriWasm sets `StackGrowth` to `Unknown`.

Verifying stack isolation. To verify stack isolation, VeriWasm checks all stack accesses of the form `mem[rsp + c]` access memory within the stack region. Specifically, for functions without a probestack, this amounts to checking:

$$-4\text{K} < \texttt{StackGrowth} + \texttt{c} < 8\text{K}.$$

[3]We're working with Bytecode Alliance to integrate this patch into Lucet.

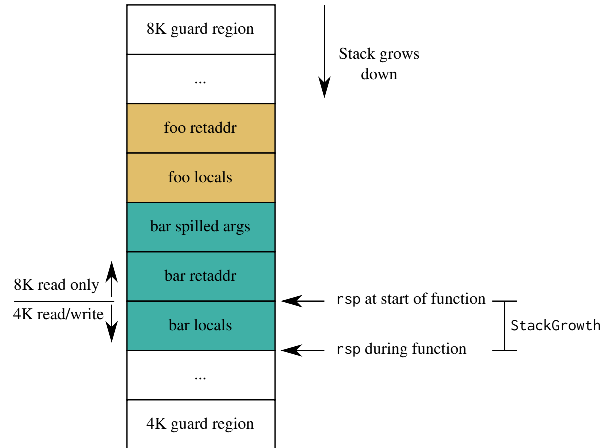

Figure 3. Stack layout of function `bar` called by function `foo`. VeriWasm ensures that `bar` can safely read and write to local variables (at most 4K) in its own stack frame but not beyond the stack frame pointer—to clobber its return address or `foo`'s stack frame. Moreover, VeriWasm ensures that `bar` can also read its spilled arguments (at most 8K).

For functions with probestack, we instead check:

$$\min(-4\text{K}, \texttt{k}) < \texttt{StackGrowth} + \texttt{c} < 8\text{K},$$

where `k` is the argument to the `probestack(k)` call.

Verifying stack frame integrity. To verify stack frame integrity, VeriWasm checks that for all stack writes of the form `mem[rsp + c] = x` fall within the 4K read-write region, i.e.,

$$-4\text{K} < \texttt{StackGrowth} + \texttt{c} < 0.$$

This ensures that the function cannot write past the stack frame pointer to, for example, clobber return addresses.

The following code shows a function `bar` which writes the value of `rdi` to a stack slot, then reads a spilled argument into the `rax` register.

```
1  bar:
2    ; TRACK: StackGrowth 0
3    push rbp;
4    ; TRACK: StackGrowth -8
5    mov rbp, rsp;
6    sub rsp, 0x10;
7    ; TRACK: StackGrowth -24
8    mov [rsp + 0x8], rdi;
9    ; ASSERT: -4K < StackGrowth + 0x8 < 0
10   mov rax, [rsp + 0x20];
11   ; ASSERT: -4K < StackGrowth + 0x20 < 8K
12   ...
```

Lines 3 and 6 decrease `StackGrowth`: the push on line 3 decreases `StackGrowth` by eight and the allocation of stack space on line 6 decreases `StackGrowth` by sixteen. On line 8, the stack write is checked to verify the frame's integrity. This check passes because `StackGrowth` must be `-24` at this point, so the address `mov rsp + 0x8` must be a local stack variable. On line 10, the stack read is checked to ensures the read is within the stack region. This check also passes because it reads from the 8K before the stack frame, accessing a spilled argument.

C. Global variable isolation

VeriWasm validates that all global variable accesses are contained within the global memory region; the region consists of one or more 4K pages, and Lucet statically declares the region's size within the binary. Lucet compiles all global variable accesses to constant offsets from the base of the global memory region. This makes validation simple: VeriWasm tracks the base of the global memory region, and validates that it is only accessed with a constant offset that is within the declared size of the region.

Tracking global variables. VeriWasm tracks which variables point to the start of the global memory region (`GlobalsBase`). Initially, Lucet stores `GlobalsBase` just below the base of the linear memory, at `LinearMemBase - 0x10`. VeriWasm figures out which variables point to `GlobalBase` similarly to how linear memory safety analysis tracks `LinearMemBase` (§IV-A).

Verifying global variable safety. VeriWasm checks that all global variables accesses are contained within the global variable memory region. At each global variable access, it checks that the offset is less than `GlobalSize`, the size of the globals region. The following code shows how VeriWasm validates a function that increments a global variable:

```
1  ; ASSUME GlobalSize 4096
2  ; TRACK: rax, rbx, ... are Unknown
3  ...
4  mov rax, [rdi - 0x10];
5  ; ASSERT: rdi is LinearMemBase
6  ; TRACK: rax GlobalsBase
7  mov rbx, [rax + 0x18];
8  ; ASSERT: rax is GlobalsBase
9  ; ASSERT: 0x18 < GlobalSize
10 add rbx, 1;
11 mov [rax + 0x18], rbx;
12 ; ASSERT: rax is GlobalsBase
13 ; ASSERT: 0x18 < GlobalSize
14 ...
```

First, line 4 loads `GlobalsBase`, and VeriWasm validates that it's loaded from the correct location; this would fail if `GlobalsBase` were loaded from anywhere besides `[LinearMemBase - 0x10]`, since anything else below `LinearMemBase` is illegal. Next, VeriWasm checks that the offset of the global variable is less than `GlobalSize`: this ensures that the access stays in the allocated region (line 7). VeriWasm similarly checks the write offset on line 11. All accesses are to offsets less than the `GlobalSize` of 4096, so the checks pass.

D. Control flow safety

VeriWasm verifies the control flow safety of native-compiled Wasm binaries. Specifically, we verify (1) *jump target validity*—that indirect jumps target valid code blocks; (2) *call target validity*—that indirect calls target the start of valid functions; and (3) *return target validity*—that function returns actually return to their call sites.

1) Jump target validity: To ensure that the control flow graph for each function is complete, VeriWasm resolves all indirect jumps within the CFG. While the problem of statically deciding the targets of computed jumps is notoriously difficult [81] (if possible at all), Wasm's language level restrictions on indirect jumps make it tractable for native-compiled Wasm modules. In particular, Wasm does not support arbitrary computed jumps. The only computed jump Wasm supports is the branch or jump table instruction `br_table`. Lucet compiles these jump table instructions as follows:

```
1  ; TRACK: rax, rbx, ... are Unknown
2  ...
3  cmp rax, 0x7; compare jump index to table size
4  jae default_case;
5  ; TRACK: rax is Checked(0x7)
6  mov rdx, 0x4000; load of a jump table
7  ; TRACK: rdx is JumpTableBase(0x4000)
8  mov rbx, [rdx + rax * 4];
9  ; TRACK: rbx is JumpOffset
10 add rdx, rbx; jump to the target
11 ; TRACK: rdx is JumpTarget
12 jmp rdx;
```

Here, the native-compiled Wasm first checks that the jump index (`rax`) is less than the total number of table entries (lines 3 and 4). If it is, the code performs a lookup into the jump table and jumps to the appropriate case (lines 6-12).

VeriWasm exploits the structure of the jump table check to determine all possible jump targets for each jump table.[4] Then, VeriWasm verifies that every jump target is safe by running the verifier on the CFG at the target site. (This is an overapproximation of possible jump targets, since some jump table targets may remain unused.) VeriWasm does this using a fix-point algorithm: while a single pass of this analysis may resolve all indirect jumps with respect to a particular CFG, resolving these jumps may also reveal additional indirect jumps which require further disassembly. Because of this interdependence between the CFG and the indirect jump analysis, VeriWasm alternates between disassembling passes and jump-resolution passes. It stops when it reaches a fixed point in the CFG generation process, i.e., when the analysis no longer resolves new jumps.

Tracking jump targets. To verify that indirect jump targets are safe, VeriWasm must identify all potential jump targets. To this end, VeriWasm tracks which variables within a function represent intermediate steps of a valid indirect jump target lookup. On line 6 in the above code snippet, for example, `rdx` takes on the abstract value `JumpTableBase(0x4000)` since it was assigned the constant `0x4000`. In this same block, `rax` has the abstract value `Checked(0x7)`; it was checked to be less than `0x7` on lines 3 and 4.

Verifying jump targets. VeriWasm uses the abstract values from the analysis phase to identify all possible targets for the indirect jump on line 12. In particular, VeriWasm uses the jump table bounds check—the abstract `Checked(·)` value—to identify the number of jump target entries from the jump table base—the abstract `JumpTableBase(·)` value—it needs to verify. For example, in the above code, VeriWasm identifies that the possible targets are the first seven entries at `0x4000`, and then disassembles and verifies the code at those locations.

[4] If VeriWasm encounters a computed jump that does not follow this pattern, the code is not a valid Lucet compilation output and the safety check fails.

2) Call target validity: To verify that a function is safe, VeriWasm validates that every function that it calls is also safe. Indirect calls make this hard: VeriWasm cannot necessarily determine the target of the call. Instead, VeriWasm overapproximates the set of possible targets of the indirect call, and validates that every possible target is safe. VeriWasm can do this because Wasm treats indirect calls as a lookup into a statically known function pointer table.

Lucet performs a dynamic safety check on all indirect calls to ensure each call target is present in the indirect function table. By verifying that all indirect calls enforce this check, and that all function pointers in the module indirect-function table point to the start of verified functions, we inductively verify that indirect calls are safe.

While Wasm requires all indirect calls to be well-typed—and Lucet enforces this with a runtime check—VeriWasm does not attempt to verify this. Doing so would be hard. Computing the number of arguments to each function alone is difficult—it requires us to infer the number of arguments passed at each call site. This is harder once we consider indirect calls to the function—to infer the number of arguments passed to a function we would have to analyze all indirect call sites that could potentially call the function. Happily, calling a function with an incorrect type for Lucet compiled Wasm does not affect SFI-safety. First, Lucet validates arguments before they are used in SFI-sensitive operations (e.g., loads and stores), regardless of their type. Second, the 8K guard region below the stack region (§IV-B) ensures that reading stack-spilled arguments can never result in stack accesses outside the stack region.

Below we give an example of a function `baz` correctly loading a function pointer from the indirect function table and calling that function.

```
1  ; TRACK: rax, rbx, ... are Unknown
2  baz:
3    push rbp;
4    mov rpb, rsp;
5    mov r9, 0x2000;
6    ; TRACK: r9 MetadataBase
7    mov r9, [r9 + 0x8];
8    ; ASSERT r9 is MetadataBase
9    ; TRACK: r9 TableSize
10   cmp r9, rax;
11   jb case2;
12
13 case1:
14   ud2;
15
16 case2:
17   mov r9, 0x3000; base of indirect func table
18   ; TRACK: rax Bounded
19   ; TRACK: r9 FuncTableBase
20   shl rax, 0x4;
21   ; TRACK: rax PtrOffset
22   mov rax, [r9 + rax + 0x8];
23   ; ASSERT: r9 is FunctionTable
24   ; ASSERT: rax is PtrOffset
25   ; TRACK: rax FnPtr
26   call rax;
27   ; ASSERT rax is FnPtr
28   ret;
```

Lines 5-11 load the indirect function table size from the module metadata and check that `rax` is less than the total number of entries in the table, and thus that `rax` holds a valid index. This check ensures that loading the entry corresponding to the function index is in-bounds and cannot be used to read memory beyond the module boundary. In particular, if the index in `rax` is out-of-bounds (greater than the number of entries in the indirect function table) the code proceeds to `case1` which triggers an illegal instruction exception. If the index is in-bounds, lines 17-26 use the checked `rax` to lookup a function pointer in the indirect function table, and then call it. VeriWasm's abstract analysis tracks the steps of this process to ensure that all steps have been performed correctly.

Tracking call targets. The call safety analysis tracks which variables within a function are used as intermediate values in the function pointer checking process. The intermediate values of the call check are `MetadataBase` (`r9` on line 5), `TableSize` (`r9` on line 7), `Bounded` (`rax` on line 17), `PtrOffset` (`rax` after line 20) and FnPtr (`rax` after line 22). If VeriWasm cannot verify that a variable is any of these intermediate values, then the value is `Unknown` and should not be used to create a valid function pointer. If any of the operands to the two memory operations (table size lookup and function table lookup) are `Unknown`, VeriWasm triggers a safety violation, since this access could potentially break isolation. The analysis is designed in such a way that the abstract value representing a step of the lookup process can only be produced by the correct operation applied to the results of two correctly computed previous steps. For example, the abstract value `FnPtr` representing a valid function pointer can only be produced by dereferencing the base of the indirect table added to a valid entry offset in that table, which in turn must have already been bounds checked to be within the table.

Checking call check integrity. Checking call safety is as simple as checking that the register acting as the target for each indirect call has the abstract value `FnPtr`, representing a properly computed call target. This check on line 26 succeeds because `rax` has been computed through the correct sequence of steps. VeriWasm would have triggered a safety violation if the value of `rax` was not checked, or if the lookup was not performed correctly (e.g., if the pointer was modified after the code loaded it from the table).

Checking function table integrity. While validating our call check integrity ensures that the target of each indirect call check is properly loaded from the indirect call table, VeriWasm also validates that every function pointer present in the indirect call table points to the start of a valid function. VeriWasm checks that the indirect function table is located in a read-only section, and performs a one-time check when the module is loaded that all pointers present in the table target the start of validated functions.

3) Return target validity: VeriWasm validates that all functions return to their call site. To do this, VeriWasm validates that at each return site in the function the stack pointer points to the return address pushed by the calling function. We do this by verifying that the stack growth—the difference between the `rsp` and `rbp`—of Section IV is zero at each return site.

E. Making VeriWasm robust to compiler optimizations

Lucet performs optimizations after it inserts safety checks, which means that optimizations can modify these checks. Lucet can, for example, split checks across basic blocks, modify checks, or reorder checks. If VeriWasm is not precise enough, it can falsely label these modifications unsafe; below, we describe how VeriWasm handles optimizations without triggering false positives.

VeriWasm allows safety check modifications as long as all necessary checks have been performed by the time the checked variable is used in an SFI-sensitive operation (e.g., a load or a jump). One example of a safety check modification is how Lucet reorders the steps of indirect call checks. Normally, it generates indirect call checks that load the function pointer in three steps:

1) Check the function index lies within the call table.
2) Create the `PtrOffset` into the indirection function table for this index. This `PtrOffset` points to a valid entry in the call table, since the function index has already been checked.
3) Dereference the `PtrOffset` to retrieve the function pointer.

Lucet's optimizations often reorder steps one and two of the check:

1) Create a `PtrOffset` from a potentially unsafe index. This `PtrOffset` points to a valid entry in the indirect function table *only if the base index is shown to be safe before* `PtrOffset` *is dereferenced.*
2) Check the function index lies within the call table.
3) Dereference the `PtrOffset` to retrieve the function pointer.

This reordered check is safe because the function index has been checked (and therefore the `PtrOffset` must be valid) before the `PtrOffset` generated from the index is dereferenced. Crucially, it's not possible to validate this check's safety using the strict formulation in Section IV-D2, since the strict safety check requires pointer offsets to be generated from a known-valid base index. Therefore, the naive version of VeriWasm triggers false positives.

To make VeriWasm's analyses robust to reordering, we use *dependent abstract variables* (DAV). A DAV is a variable whose safety depends on the safety of another variable. VeriWasm uses DAVs in both call safety analysis and indirect jump analysis. In the example above, VeriWasm uses a `DependentPtrOffset` instead of a `PtrOffset`, since the base index has not yet been checked. If the `DependentPtrOffset` is used before its base index has been checked, validation will fail. Concretely, consider the following (correct) function that performs an indirect call with a reordered safety check:

```
1  ; TRACK: rax, rbx, ... are Unknown
2  baz:
3    push rbp;
4    mov rpb, rsp;
5    mov r9, 0x2000;
6    ; TRACK: r9 MetadataBase
7    mov r9, [r9 + 0x8];
8    ; ASSERT: r9 is MetadataBase
9    ; TRACK: r9 TableSize
10   shl rax, rdx, 0x4;
11   ; TRACK: rax DependentPtrOffset(rdx,6)
12   cmp r9, rdx;
13   jb case2;
14
15 case1:
16   ud2;
17
18 case2:
19   mov r9, 0x3000;
20   ; TRACK: rdx Bounded
21   ; TRACK: rax PtrOffset
22   ; TRACK: r9 FuncTableBase
23   mov rax, [r9 + rax + 0x8];
24   ; ASSERT: r9 is FunctionTable
25   ; ASSERT: rax is PtrOffset
26   ; TRACK: rax FnPtr
27   call rax;
28   ; ASSERT: rax is FnPtr
29   ret;
```

On line 10 `rax` is a `DependentPtrOffset` that is dependent upon the base index (`rdx`). If `rax` were immediately used to lookup the indirect function pointer, validation would fail. However, if `rdx` were checked first (as on lines 12 and 13), validation would succeed, since `rax` must necessarily be valid by the time it is used.

VeriWasm uses reaching definitions [5] to resolve DAV constraints. Reaching definition analysis records the set of locations that could have written to registers and stack locations at every point in the program as well as the expression assigned to these locations. If two registers or stack slots have identical reaching definitions, they must necessarily have the same value i.e. they are aliased. This means that if a register or stack slot is declared safe, all other variables with the same set of reaching definitions are safe as well. Whenever a variable is checked, VeriWasm also resolves all dependent abstract values with the same set of reaching definitions as the variable being checked (and any variables derived from them).

V. Verifying the Analysis

We use the Coq proof assistant to formally verify our analysis. In particular, we prove that VeriWasm is *sound*: the verifier is sound when every program it labels as safe indeed does not break SFI-safety when run. But intuitively, soundness is only useful if the verifier is sufficiently *precise*: the verifier is precise "enough" if it doesn't falsely label many programs as unsafe. In Section VI we empirically evaluate the precision of VeriWasm; in this section we focus on soundness.

To verify soundness, we first use Coq to formalize:

- An intermediate language (IL) called w64, which captures the subset of x86-64 generated by Lucet.
- The verifier itself (over w64).

Using these definitions, we then state our soundness theorem and prove it in Coq. We only verify the analysis passes; we do not verify the disassembly, control-flow graph creation, or the translation from x86-64 to w64 (§VII). We do this because the

```
⟨reg⟩    ⊨  rax | rbx | rcx | ...
⟨op⟩     ⊨  add | sub | mult | ...
⟨cond⟩   ⊨  eq | neq | lt | ...
⟨instr⟩  ⊨  reg ← LinearMem[reg + reg + reg] |
            LinearMem[reg + reg + reg] ← reg |
            LinearMemCheck reg | CallCheck reg |
            StackExpandCheck nat |
            reg ← GetGlobalsBase reg |
            reg ← Globals[i] |
            Globals[i] ← reg |
            reg ← reg | reg ← nat |
            StackExpand nat | StackContract nat |
            reg ← Stack[nat] | Stack[nat] ← reg |
            list reg ← op list reg |
            Branch cond nat nat | Jmp nat |
            ICall reg |  Call string  | Ret
```

Figure 4. Syntax for w64 intermediate language. w64 is expressive enough to reason about behavior of all natively-compiled Wasm code.

analysis pass, unlike, say, disassembly and CFG creation, is complex and has not been formalized in prior work. Moreover, exhaustively testing the analysis pass is hard—indeed, despite our best effort testing the analysis, verification revealed edge-cases we missed in early versions of the tool.

w64 intermediate language. Instead of formalizing hundreds of low-level x86-64 instructions, we create an intermediate language, w64, consisting of nineteen instructions (Figure 4). Though we cannot model arbitrary x86-64 programs using w64, w64 is expressive enough to reason about all x86-64 programs produced by Lucet.

w64 instructions capture and abstract all the relevant semantics of Lucet-generated x86-64 needed to prove SFI-safety. For example, w64 models Lucet's indirect call checking using a single `CallCheck` instruction; this abstracts over the actual x86-64 code that Lucet generates, which consists of roughly eight x86-64 instructions (§IV-D2). We similarly abstract x86-64 memory accesses into instructions that explicitly manipulate the stack, global variables, and linear memory. For example, we model any loads that use `rsp` (e.g., `mov rax, [rsp + 0x4]`) as stack-slot reads (e.g., `rax ← Stack[0x4]`), since Lucet only uses the `rsp` register as such.

w64 semantics. We formalize the w64 language using small-step operational semantics. Small-step operational semantics describe a language by specifying how expressions are evaluated (e.g., how to handle pointer dereferences) and how the execution of an instruction transforms the computational state (e.g., registers and memory).

Our semantics model the subset of x86-64 semantics necessary to prove SFI safety for Lucet-generated programs. To this end, we define a step relation $\rightarrow$ that operates on program states, consisting of (1) an instruction stream *is* and (2) a runtime environment σ, which encapsulates the stack, global variables, linear memory, indirect function table, and various runtime metadata. The relation $\langle is, \sigma\rangle \rightarrow \langle is', \sigma'\rangle$ states that $\langle is', \sigma'\rangle$ is the result of executing the first instruction of the stream *is* in the environment σ (where σ' is the resulting environment, and is' is the resulting instruction stream). Following [10], our semantics are *defensive* and explicitly leave the behavior of all SFI-breaking states unspecified. In other words, programs (starting from initial state σ_0) that violate SFI get "stuck", i.e., they reach a state from where they can no longer take a step. This means that to prove that our verifier is sound, we just need to show that any programs it deems safe *can make progress*, i.e., it does not get stuck. We do this next.

Formalizing the verifier. We formalize the verifier in Coq as a total function *verify* from a disassembled program p (represented as a list of control-flow graphs of w64 instructions) to a binary value: **safe** or **unsafe**. The verifier implements the abstract interpretation analysis of Section IV. We treat every function in the program as a potentially valid entry point for the execution—and, so, the verifier statically analyzes each function from the initial state σ_0.

Proving the verifier's soundness. Using the Coq formalization of the w64 language (the $\rightarrow$ step relation) and verifier (the *verify* function), we want to show that if the verifier deems a program safe, then executing the program indeed doesn't break SFI-safety. As stated previously, we set up our semantics so that proving that a program does not break SFI-safety amounts to showing that the program does not get stuck. Hence, we simply need to prove that:

$$(\mathit{verify}(p) = \textbf{safe}) \Rightarrow \text{never-gets-stuck}(p)$$

Never getting stuck is equivalent to being able to take a single step after taking an arbitrary number of steps from the start of the execution. We formalize this using the multi-step relation $\rightarrow^*$, the reflexive and transitive closure of the single-step relation $\rightarrow$, i.e., we say $\langle is, \sigma\rangle$ reduces to $\langle is', \sigma'\rangle$ in zero or more steps if $\langle is, \sigma\rangle \rightarrow^* \langle is', \sigma'\rangle$. Specifically, we define the soundness property as:

$$(\mathit{verify}(p) = \textbf{safe}) \Rightarrow \left[\begin{array}{l} f \in p \Rightarrow \\ \quad \left(\begin{array}{l} \langle start(f), \sigma_0\rangle \rightarrow^* \langle is, \sigma\rangle \Rightarrow \\ \exists is', \sigma', \langle is, \sigma\rangle \rightarrow \langle is', \sigma'\rangle \end{array} \right) \end{array} \right]$$

Here, all free variables are universally quantified on the outside, f ranges over all control-flow graphs in p, and $start(f)$ is the entry point of a given control-flow graph.

From the property above, it is trivial to show a *global-safety property*, i.e., if the verifier deems a program safe, the program doesn't break memory isolation during execution. Breaking memory isolation is only a subset of the behavior prohibited by the small-step semantics, so breaking memory isolation at any point during execution results in the program getting stuck. Therefore, the soundness property above is strong enough to prove the desired global-safety property.

Mechanization effort. We mechanize the proof of the soundness property in Coq. Our mechanized verification efforts consists of 1800 lines of Coq, 500 of which are devoted to proofs. The Coq source is open source and available online [29].

Verification results. Our verification effort helped us identify *serious* flaws in the early VeriWasm implementation. Several times, we had oversimplified the behavior of x86-64, and as a result forgot to add checks for unsafe behavior. Specifically, we forgot to:

- Ensure that `rdi` is set to linear memory base before direct and indirect function calls. This is important because the verifier assumes that `rdi` holds the linear memory base value at the beginning of functions.
- Clear the abstract safety information of callee-saved register after calls. Functions may not respect calling conventions, so callee-saved registers aren't guaranteed to be restored appropriately after a call.
- Limit lookups in the caller's stack frame to 8KB of stack memory. Since parameters are located in the caller's stack frame in native-compiled Wasm, callees must have access to the caller's stack frame—but only up to 8KB.
- Ensure a stack expansion of more than 4KB is preceded by a *probestack* call. This ensures a module cannot request a large amount of stack space and skip past a guard page.

Missing any of these edge cases leads to an unsound verifier.

VI. Evaluation

We evaluate VeriWasm by asking three questions:

- Does VeriWasm find SFI breaking bugs—can it discover compilation bugs in Wasm binaries that allow accesses outside sandbox memory?
- Does VeriWasm have a low false positive rate—does it avoid incorrectly flagging Wasm binaries that are actually safe?
- Is VeriWasm fast enough—can it validate Wasm modules as part of the compilation process for browsers and edge cloud providers?

To check if VeriWasm is able to find SFI breaking bugs, we evaluate it on a suite of known bugs from previous SFI systems and Wasm compilers (§VI-A); it's able to identify every bug in this suite. We also set up a fuzz harness for Lucet; while this did not reveal any bugs on the current version of Lucet, it helped us eliminate false positives in the VeriWasm tool. To evaluate VeriWasm's false positive rate and performance, we validate four sets of benchmarks—SPEC2006, the Lucet compiler's Shootout microbenchmark suite, two Wasm sandboxed libraries shipped by the Firefox browser, and 103 Wasm modules from the edge-cloud provider Fastly for a total of 119 Wasm modules. VeriWasm reports no false positives and validates most of these applications in less than twenty seconds; this latency, while not sufficient for just-in-time applications, allows Firefox and Fastly to run VeriWasm on nightly re-builds of Wasm binaries.

Experimental setup. We run all experiments (with the exception of verifying Fastly's binaries) on a 2.1GHz Intel Xeon Platinum 8160 machine with 96 cores and 1 TB of RAM running Arch Linux 5.8.14. We evaluate the Fastly modules on a 5.5GHz Intel quad core i7-8559U machine with 8GB of RAM. All experiments run on a single core, and no experiment uses more than 6GB of RAM. We compile the SPEC2006 and Shootout benchmarks using the Clang compiler (version 10.0.0) to go from C/C++ to Wasm, then compile the results to x86-64 using the Lucet compiler (version 0.7). The Firefox Wasm sandboxed libraries come from the Firefox nightly build (version 78.0a1 on May 4th, 2020). Fastly customers compiled their applications from the source language to Wasm, and Fastly compiles from Wasm to x86-64 on their own servers using the Lucet compiler (version 0.7).

Implementation. We implement VeriWasm in ≈3000 lines of Rust. The formal verification (§V) consists of ≈1800 lines of Coq, ≈500 lines of which are dedicated to proofs. VeriWasm uses the Yaxpeax disassembler for disassembly [3].

A. Does VeriWasm find SFI breaking bugs

Testing. We test if VeriWasm finds SFI-breaking errors by creating a suite of 11 bugs from other SFI toolchains like NaCl [84], miSFIt [66], and PittSFIeld [45], as well as old bugs from the Lucet compiler. These bugs fall into different categories: two violate call safety, three violate stack safety, two violate linear memory safety, one violates jump safety, and three use illegal instructions. The most interesting bugs include:

- A stack out-of-bounds write where the SFI scheme does not prevent the stack pointer from being moved outside the allocated stack range. This bug was found in the MiSFIt [66] SFI system by McCamant et al. [45]. Malicious code can exploit the bug by repeatedly allocating stack space until the stack pointer points to a different memory region (e.g., a different module's address space). VeriWasm catches this bug by verifying stack isolation and stack frame integrity.
- An unchecked memory access during indirect jump target lookup. A Lucet optimization re-ordered instructions so that bounds checks for indirect jump indices occurred after jump table lookups. VeriWasm catches this bug as part of its control flow safety checking, which ensures that jump table lookups only occur after the indirect jump index has been bounds checked.
- A bug where the memory safety checks and the control flow safety checks are accidentally mixed up, allowing for control flow through a register which has only been checked to be safe for memory access. Unlike the previous bugs in SFI compilers, this bug was discovered in the *verifier* for the PittSFIeld SFI scheme by Kroll et. al [31]. Specifically, the verifier marked certain binaries as safe, when they were actually unsafe. VeriWasm successfully classifies binaries with such unsafe pattern as unsafe.

We run VeriWasm on code that uncovers a given vulnerability by by either (1) compiling proof-of-concept code from an original bug report or (2) handwriting assembly with the vulnerability (when the proof-of-concept is not available). In some cases, we have to translate the bug to Wasm; for example, one bug unsafely manipulates a function pointer that has already been checked by NaCl. Since Lucet uses a different kind of

	astar	bzip2	gobmk	h264ref	lbm	libquantum	mcf	milc	namd	povray	sjeng	soplex	sphinx
Average Function Validation Time (s)	0.02	0.05	0.01	0.02	0.08	0.05	0.04	0.02	0.04	0.01	0.05	0.0	0.02
Max Function Validation Time (s)	6.23	5.82	5.63	5.78	10.59	10.55	6.08	6.42	5.73	5.59	9.59	10.14	9.91
# Functions in Module	334	170	2638	695	142	233	153	394	363	1901	279	3895	490
Total Validation Time (s)	7.27	7.74	13.6	16.57	11.83	10.78	6.72	7.17	15.66	19.31	13.83	19.25	11.01

Figure 5. Average function verification times, max function verification times, and total module verification times of SPEC2006 applications.

	shootout
Average Function Validation Time (s)	0.63
Max Function Validation Time (s)	71.45
# Functions in Module	124
Total Validation Time (s)	78.04

Figure 6. Average function verification times, max function verification times, and total module verification times of Lucet's microbenchmarks.

	libgraphite	libogg
Average Function Validation Time (s)	0.01	0.0
Max Function Validation Time (s)	7.61	0.03
# Functions in Module	674	270
Total Validation Time (s)	7.89	0.12

Figure 7. Average function verification times, max function verification times, and total module verification times of Firefox libraries currently deployed as native-compiled Wasm.

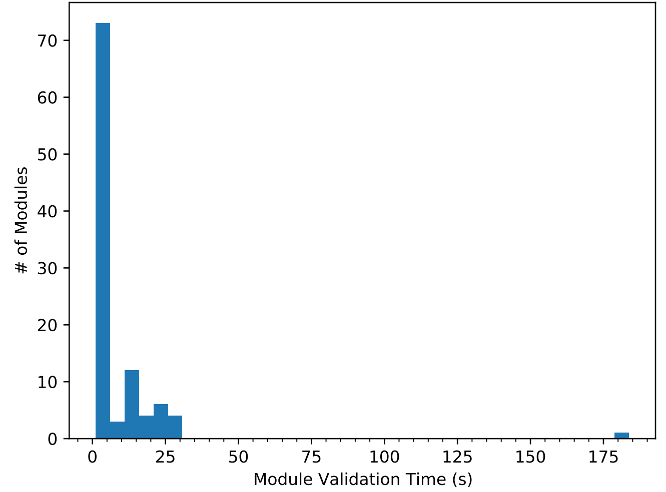

Figure 8. Total verification time for each of Fastly's client applications

function pointer check based on an indirect function table, we modify this bug use Lucet's function pointer check.

Fuzzing. We also integrated VeriWasm into a fuzz testing pipeline. Specifically, we (1) use Csmith [82] to randomly generate valid C/C++ programs, then compile them to Wasm with Clang and (2) directly generate random Wasm modules using Binaryen [2]. We then compile these Wasm modules with Lucet and run VeriWasm on the resulting binaries. We run the fuzzing infrastructure on over 2 million Csmith-generated programs and 20 million Binaryen-generated programs, but do not find bugs in current version of Lucet.

The fuzzing infrastructure did, however, reveal early bugs in VeriWasm—VeriWasm declared some safe programs unsafe because Lucet optimizations reordered and manipulated some dynamic safety checks in a safe but unexpected way. We used these results to improve VeriWasm's precision (see §IV-E).

B. Can VeriWasm validate correct programs?

We formally prove that VeriWasm cannot classify an unsafe program as safe (§V); still, VeriWasm *can* flag safe code as unsafe. This may happen, say, if Lucet runs an optimization that VeriWasm cannot precisely reason about (e.g., the range-based loop check hoisting optimization Zeng et al. describe [87]). To check if VeriWasm's analysis is precise enough to avoid false positives on real code, we test it on binaries from four sources.

VeriWasm validates all four benchmark sets—a total of 119 Wasm binaries from SPEC2006 [27], Lucet's Shootout microbenchmarks, Wasm sandboxed libraries in Firefox [49], and Wasm modules deployed in Fastly's edge cloud—with no false positives. SPEC2006 presents realistic workloads on applications like video compression, speech recognition, and ray tracing. We run VeriWasm on thirteen of nineteen SPEC benchmarks (all written in C and C++) and, like [28], we exclude six because they contain constructs that aren't representable in Wasm (e.g., `longjmp`, exceptions, and unsupported system calls). Shootout, which Lucet uses for performance testing, consists of a single Wasm module containing benchmarks like matrix multiplication and AES encryption; several benchmarks are particularly interesting for VeriWasm, since they contain some uncommon code patterns (e.g., very large switch tables). We also evaluate VeriWasm on two Wasm sandboxed libraries that ship with Firefox nightly [50], and 103 Wasm modules currently deployed by Fastly [55]. Both of these benchmarks are of deployed real-world applications of Wasm sandboxing.

C. Is VeriWasm fast enough?

To evaluate if VeriWasm is fast enough to run on Firefox and Fastly's nightly builds of Wasm modules we use the four sources of benchmarks from the previous section. Verification of real modules takes on average 8.6 seconds and a median time of 1.7 seconds; while this is sufficiently fast for our intended use case of validation before a nightly build, it is too slow for low latency applications like just-in-time compilers.

SPEC2006 performance. VeriWasm takes between 6.7 and 19.4 seconds to validate each SPEC module (Figure 5). On average, the top 1% of functions account for 87% of execution time. We observe this for all benchmarks: a few large complex functions use most of the total verification time.

Shootout performance. VeriWasm requires 78 seconds to validate the Shootout module. A single function—a function that contains a huge switch table with 4096 cases (see Figure 6)—dominates the verification time: it take takes 71 seconds to validate. When compiling this function, Lucet uses separate local variables for each switch case. This results in code with over 10,000 local variables that are maintained on the stack. So, VeriWasm takes a long time to validate this code—it must track information about all of these variables across a large, complex function.

Firefox performance. VeriWasm requires between 0.12 and 7.9 seconds to validate the Firefox libraries (see Figure 7). This overhead is reasonable for our use case—verifying nightly builds—even with many libraries. Indeed, this is even cheap enough to run in the Firefox continuous integration tool.

Fastly client performance. VeriWasm takes between 1 and 183.7 seconds to verify each Fastly Wasm application, The median verification time is 1.85 seconds (see Figure 8). These applications have between 501 and 3123 functions, with a median of 621. Because the Fastly application code is confidential, we unfortunately can't diagnose and thus report why one of the applications takes three minutes to verify.

Performance breakdown. For each of our 119 modules, VeriWasm spends, on average, 75.2% of the verification time performing (CFG generation and) indirect jump analysis; 21.9% checking call safety; and, 2.9% verifying stack and heap safety. Indirect jump analysis is more expensive than other analyses for two reasons. First, this analysis records the reaching definitions of each value in the program in case they are needed; the other analyses only track values that have been computed by code that resembles safety checks. Second, because the analysis is inherently coupled with CFG generation (§IV-D1), VeriWasm sometimes needs need to perform the jump analysis multiple times to resolve all the jumps in a particular function CFG.

When verifying a module, we find that VeriWasm spends, on average, 70.8% of its time verifying roughly 1% of the functions in the module. These functions are large and complex—e.g., the median number of basic blocks in one of these functions is 521, which is roughly $47\times$ larger than the median number of blocks of function across all the modules we verify (11).

VII. Limitations and Future Work

Unverified disassembly. VeriWasm relies on an unverified disassembler; bugs here can lead to bugs in VeriWasm's verification. However, this is not a fundamental limitation of the technique since verified disassembly has been demonstrated in prior work [47] and VeriWasm can be modified to adopt such approaches.

VeriWasm's speed. Although VeriWasm is sufficiently fast for its intended use case of checking nightly builds of Wasm code in browsers and CDN deployments, it is currently not fast enough to perform online verification in low latency applications like Wasm JIT compilers. This is because VeriWasm's control flow analyses seem to scale worse to large, complex functions than its other analyses (as discussed in VeriWasm VI-C). To address this issue and enable online verification in the future, we plan to: (1) optimize the performance of these control flow analyses, and (2) work with the Lucet team to make Lucet generated code more easily verifiable without compromising on performance.

Overfitting to Lucet. VeriWasm currently only checks x86-64 code generated by the Lucet Wasm compiler. While we have not extended VeriWasm to other platforms or compilers, we designed VeriWasm to be extensible and general: our verifier takes advantage of Wasm properties instead of Lucet's compilation of Wasm whenever possible. For example, VeriWasm verifies indirect calls by checking that the target of the call is loaded from the indirect function table. The indirect function table is a Wasm concept, not a Lucet detail. But, the exact layout of the indirect call table (in native code) is compiler-specific and we would have to extend VeriWasm to accommodate layouts that differ from Lucet.

We would similarly have to extend VeriWasm to account for potentially more complex optimization passes that affect SFI checks. While Lucet's optimizations of SFI safety checks are the most advanced today, both Lucet and other Wasm compilers may implement more complex SFI optimizations in the future (e.g., following [87]) We are working to integrate VeriWasm and verification into the Wasm ecosystem and hope to co-evolve the verifier with the bytecode.

Unverified Wasm runtime. Wasm modules require a Wasm runtime that creates and manages memory regions for the module as well as provides secure access to syscalls. VeriWasm does not verify any part of this runtime and bugs in the runtime are out of scope of this work. However, bugs in SFI runtimes have been found in the past [51] and their verification remains an open problem.

VIII. Related Work

In this section, we put VeriWasm in context of related work.

SFI system and verifier safety. Before Wasm, most SFI systems [80], [45], [84], [62] were designed with binary validation in mind and came packaged with the corresponding verifier tools. These systems did not allow optimizations to move or eliminate SFI checks, allowing for extremely compact verifiers that can ensure binary safety by analyzing just a few instructions at a time [45]. Rocksalt [47] implements a fully verified version of such an SFI verifier, verifying both x86 disassembly and correctness of the analysis of Native Client [84] binaries. Tools like ERIM [76] that enforce fault isolation via hardware features can simplify validation more; for instance, the ERIM's verifier only has to check that the sandboxed component never uses byte sequences that decode to privileged hardware instructions that disable sandboxing. Unlike these tools, VeriWasm validates natively-compiled Wasm modules where optimizations may have eliminated and moved security checks in order to improve performance, making validation more challenging. Thus, the VeriWasm verifier checks for security properties that are comparatively more complex (§III) at a per-function level, and relies on a verified correct analysis (§V) to provide a safe verifier; unlike RockSalt, VeriWasm does not verify correct x86 disassembly.

Other SFI systems also use schemes that require more complex validation. For instance, Strato [86] allows optimizations on SFI security checks, and provides a verifier that uses range analysis to ensure that pointers are correctly bounds checked

(an approach other verifiers also use [87]). While VeriWasm employs a similar analysis (§IV-A), it must also account for Wasm's trusted stack; for instance, the example in Section IV-E shows how VeriWasm handles a case where security checks are elided when loading multiple times from the same location in the trusted stack.

Besson et al. focus on validating SFI binaries with a trusted stack [10]. They model the semantics of and implement an SFI verifier using an abstract interpretation to validate stack and heap safety, an approach VeriWasm also follows. In addition, VeriWasm identifies and addresses several more requirements necessary for Wasm verification; in particular, VeriWasm validates in the presence of indirect calls and jumps, compiler optimizations, and dynamic values for heap-base address etc. by leveraging knowledge about Wasm and Wasm compilers.

Necula et al. present a more general approach for verifying complex security properties called proof carrying code [52]. In this approach, compilers include a proof of safety along with a binary. While this allows validation of complex security properties, VeriWasm must work with the existing eco-system of Wasm modules, which does not include safety proofs.

Tan [70] provides a more detailed history and analysis of earlier SFI tools and their validation toolchains.

Compilation and translation safety. An alternate approach to ensuring safety is to verify the compiler itself. For instance, Kroll et al. [32] implement a verified SFI system on top of the Compcert verified compiler [40], and there is ongoing work to build a verified Wasm compiler [13]. More generally, CompCert and its variations [39], [79], [68] and CakeML [33] are examples of foundationally verified compilers, and there are many examples of verified optimizations for these compilers [48], [7].

Translation validation [60], [56] is a different approach to compiler correctness; it proves that some property—most extremely, equivalence—holds before and after compiler optimizations. There's translation validation work for optimization passes in industrial compilers like gcc [53] and LLVM [71], [72], [67]; there's also work on formally verified translation validation for CompCert in Coq [59], [73], [74], [75].

Another approach is to provide a domain-specific language for writing verified compiler optimizations [43], [37], [38], [34], [14]; this doesn't verify the whole compiler, but verifies that a single optimization (or other) pass is correct.

The DSL approach can verify that optimization passes are correct, while translation validation can verify that a single compilation of a given program is correct; whole-compiler verification offers the strongest guarentees of end-to-end correctness. VeriWasm's approach trades-off completeness for easier proof burden and maintenance of the compiler toolchain. Specifically, since VeriWasm only verifies the verifier—a much simpler tool than a full compiler or optimization pass—it has a comparatively smaller proof burden. Futhermore, VeriWasm's approach allows changes to the compiler and optimization passes without changes (or with minor changes) to the verifier or the verifier proofs.

Retrofitting security checks. VeriWasm's analysis techniques are similar to some tools that retrofit security checks in binaries. For instance, ARMor [89] analyzes and rewrites ARM binaries to ensure SFI which is verified via abstract interpretation; Zhang et al.[88] analyze and modify binary executables to apply control flow integrity without source code; StackArmor [17] rewrites the stack usage in binaries to prevent stack based attacks by enforcing the safe stack policy [35]. Unlike these systems, which may conservatively add checks when analyses proves too complicated or does not terminate, VeriWasm operates without modifying the generated assembly. Additionally, VeriWasm leverages properties of Wasm to further simplify analysis—for instance, to ensure safety even in the presence of indirect jumps.

Abstract interpretation. Abstract interpretation [20] has been verifying program properties and finding bugs for over forty years. The Astrée static analyzer [21] has verified absence of certain errors in space vehicles [8], and many works [58], [78] use abstract interpretation for everything from verification to synthesis. There are even verified static analysis passes and frameworks [11], [30], [12]. Like these works, VeriWasm uses a verified abstract interpretation passes but specifically focuses on showing that binaries are safely sandboxed.

Bug finding. Bug finding tools can also identify security flaws. They use techniques like symbolic execution [15], concolic execution [24], [85], [63], fuzzing [82], [1], and binary instrumentation [54]. These tools find several classes of security bugs like use-after-frees [64], race conditions [61], [65], stack overflows [58], and more; some use fast but unsound analysis to quickly find bugs with low false positives [9]. In contrast, VeriWasm cannot check for general classes of security bugs and instead only validates the Wasm security properties; it uses a sound analysis and may produce false positives (§VI).

IX. Conclusion

Complex software systems have bugs, and compilers—with their enormous attack surface—are no exception. Luckily, real-world exploits organically resulting from flaws in code generation are rare because they require byzantine inputs atypical of real programs. However, once the compiler takes on the role of protecting the execution environment from the behavior of compiled software, this dynamic is reversed—any compiler flaw is available for an attacker to exploit. This problem, in the context of Wasm, motivates our work.

Wasm offers the promise of high-performance, portable, safe code. But this promise of safety requires us to trust that the compiler inserts SFI checks in all necessary places and that these checks are correctly handled in all subsequent optimization passes. When this trust fails, so do all safety guarantees.

In this paper, we advocate an alternative approach—trust the compiler to do its job but, just in case, verify the safety properties of the native code it has compiled. We present VeriWasm, a tool to validate that Wasm binaries produced by the Lucet compiler do not have missing security checks that can break isolation. VeriWasm uses simple abstract interpretation passes to establish local per-function properties, which is sufficient to prove SFI safety of an entire Wasm binary. VeriWasm has validated over 22 million auto generated Wasm binaries with no false positives (so far), and is also able to

validate real world Wasm binaries used in Fastly's edge cloud and in the Firefox browser.

ACKNOWLEDGMENT

We thank the anonymous reviewers and our shepherd, Gang Tan, for their insightful comments. We thank Nick Fitzgerald, Adam Foltzer, Jonathan Foote, Pat Hickey, Hovav Shacham, and Andy Wortman for fruitful discussions. This work was supported in part by gifts from Fastly and Cisco; by the NSF under Grant Number CCF-1918573; and by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

REFERENCES

[1] google/afl. https://github.com/google/AFL, 2013.

[2] Binaryen WebAssembly toolchain. https://github.com/WebAssembly/binaryen/, 2015.

[3] Yaxpeax-x86 disassembler. https://github.com/iximeow/yaxpeax-x86, 2020.

[4] Ali-Reza Adl-Tabatabai, Geoff Langdale, Steven Lucco, and Robert Wahbe. Efficient and language-independent mobile programs. In *PLDI*, 1996.

[5] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers: Principles, Techniques, and Tools*. Pearson Education, 1986.

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

[7] Heiko Becker, Eva Darulova, Magnus O Myreen, and Zachary Tatlock. Icing: Supporting fast-math style optimizations in a verified compiler. In *CAV*, 2019.

[8] Julien Bertrane, Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Static analysis and verification of aerospace software by abstract interpretation. In *AIAA I@A*, 2010.

[9] Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. A few billion lines of code later: Using static analysis to find bugs in the real world. In *CACM*, 2010.

[10] Frédéric Besson, Thomas Jensen, and Julien Lepiller. Modular software fault isolation as abstract interpretation. In *SAS*, 2018.

[11] Sandrine Blazy, Vincent Laporte, André Maroneze, and David Pichardie. Formal verification of a C value analysis based on abstract interpretation. In *SQAS*, 2013.

[12] Sandrine Blazy, Vincent Laporte, and David Pichardie. Verified abstract interpretation techniques for disassembling low-level self-modifying code. In *Journal of Automated Reasoning*, 2016.

[13] Jay Bosamiya, Benjamin Lim, and Bryan Parno. WebAssembly as an intermediate language for provably-safe software sandboxing. In *PriSC*, 2020.

[14] Fraser Brown, John Renner, Andres Nötzli, Sorin Lerner, Hovav Shacham, and Deian Stefan. Towards a verified range analysis for JavaScript JITs. In *PLDI*, 2020.

[15] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[16] Fiona Campbell. The portable UCSD p-system. In *MICROPRO*, 1983.

[17] Xi Chen, Asia Slowinska, Dennis Andriesse, Herbert Bos, and Cristiano Giuffrida. Stackarmor: Comprehensive protection from stack-based memory error vulnerabilities for binaries. In *NDSS*, 2015.

[18] Adam Chlipala. A verified compiler for an impure functional language. In *POPL*, 2010.

[19] Melvin E. Conway. Proposal for an UNCOL. In *CACM*, 1958.

[20] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[21] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. The ASTRÉE analyzer. In *ESOP*, 2005.

[22] Alan Donovan, Robert Muth, Brad Chen, and David Sehr. PNaCl: Portable native client executables. *Google White Paper*, 2010.

[23] Ulfar Erlingsson and Fred B Schneider. SASI enforcement of security policies: A retrospective. In *DISCEX*, 2000.

[24] Patrice Godefroid, Michael Y Levin, and David Molnar. Sage: Whitebox fuzzing for security testing. In *Queue*, 2012.

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

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

[27] John L Henning. SPEC CPU 2006 benchmark descriptions. In *SIGARCH Computer Architecture News*, 2006.

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

[29] Evan Johnson, David Thien, Yousef Alhessi, Shravan Narayan, Fraser Brown, Sorin Lerner, Tyler McMullen, Stefan Savage, and Deian Stefan. Veriwasm verifier. https://veriwasm.programming.systems, 2020.

[30] Jacques-Henri Jourdan, Vincent Laporte, Sandrine Blazy, Xavier Leroy, and David Pichardie. A formally-verified C static analyzer. In *POPL*, 2015.

[31] J Kroll and Drew Dean. BakerSFIeld: bringing software fault isolation to x64. *SRI International, Tech. Rep.*, 2009.

[32] J. A. Kroll, G. Stewart, and A. W. Appel. Portable software fault isolation. In *CSF*, 2014.

[33] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In *POPL*, 2014.

[34] Sudipta Kundu, Zachary Tatlock, and Sorin Lerner. Proving optimizations correct using parameterized program equivalence. In *PLDI*, 2009.

[35] Volodymyr Kuznetzov, László Szekeres, Mathias Payer, George Candea, R Sekar, and Dawn Song. Code-pointer integrity. In *The continuing arms race: Code-reuse attacks and defenses*. ACM, 2018.

[36] Vu Le, Chengnian Sun, and Zhendong Su. Finding deep compiler bugs via guided stochastic program mutation. In *OOPSLA*, 2015.

[37] Sorin Lerner, Todd Millstein, and Craig Chambers. Automatically proving the correctness of compiler optimizations. In *PLDI*, 2003.

[38] Sorin Lerner, Todd Millstein, Erika Rice, and Craig Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *POPL*, 2005.

[39] Xavier Leroy. Formal certification of a compiler back-end or: programming a compiler with a proof assistant. In *POPL*, 2006.

[40] Xavier Leroy. Formal verification of a realistic compiler. In *CACM*, 2009.

[41] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert - A formally verified optimizing compiler. In *ERTS*, 2016.

[42] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification*. Addison-Wesley Professional, 2014.

[43] Nuno P Lopes, David Menendez, Santosh Nagarakatte, and John Regehr. Provably correct peephole optimizations with Alive. In *PLDI*, 2015.

[44] Stavros Macrakis. The structure of ANDF: Principles and examples. *Open Software Foundation*, 1992.

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

[46] Erik Meijer and John Gough. Technical overview of the common language runtime. http://research.microsoft.com/~emeijer/papers/CLR.pdf, 2001.

[47] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. RockSalt: Better, faster, stronger SFI for the x86. In *PLDI*, 2012.

[48] Eric Mullen, Daryl Zuniga, Zachary Tatlock, and Dan Grossman. Verified peephole optimizations for CompCert. In *PLDI*, 2016.

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

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

[51] Native Client team. Native Client security contest archive. https://developers.chrome.com/native-client/community/security-contest/index, 2009.

[52] George C Necula. Proof-carrying code. In *POPL*, 1997.

[53] George C. Necula. Translation validation for an optimizing compiler. In *PLDI*, 2000.

[54] Nicholas Nethercote and Julian Seward. Valgrind: A framework for heavyweight dynamic binary instrumentation. In *PLDI*, 2007.

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

[56] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *TACAS*, 1998.

[57] John Regehr, Yang Chen, Pascal Cuoq, Eric Eide, Chucky Ellison, and Xuejun Yang. Test-case reduction for C compiler bugs. In *PLDI*, 2012.

[58] John Regehr, Alastair Reid, and Kirk Webb. Eliminating stack overflow by abstract interpretation. In *TECS*, 2005.

[59] Silvain Rideau and Xavier Leroy. Validating register allocation and spilling. In *CC/ETAPS*, 2010.

[60] Hanan Samet. *Automatically proving the correctness of translations involving optimized code*. PhD thesis, Stanford University, 1975.

[61] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multithreaded programs. In *TOCS*, 1997.

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

[63] Koushik Sen, Darko Marinov, and Gul Agha. Cute: A concolic unit testing engine for C. In *ESEC/FSE*, 2005.

[64] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *USENIX ATC*, 2012.

[65] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data race detection in practice. In *WBIA*, 2009.

[66] Christopher Small and Margo Seltzer. MiSFIT: constructing safe extensible systems. In *IEEE Concurrency*, 1998.

[67] Michael Stepp, Ross Tate, and Sorin Lerner. Equality-based translation validator for LLVM. In *CAV*, 2011.

[68] Gordon Stewart, Lennart Beringer, Santiago Cuellar, and Andrew W. Appel. Compositional CompCert. In *POPL*, 2015.

[69] Chengnian Sun, Vu Le, and Zhendong Su. Finding compiler bugs via live code mutation. In *OOPSLA*, 2016.

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

[71] Jubi Taneja, Zhengyang Liu, and John Regehr. Testing static analyses for precision and soundness. In *CGO*, 2020.

[72] Jean-Baptiste Tristan, Paul Govereau, and Greg Morrisett. Evaluating value-graph translation validation for LLVM. In *PLDI*, 2011.

[73] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *POPL*, 2008.

[74] Jean-Baptiste Tristan and Xavier Leroy. Verified validation of lazy code motion. In *PLDI*, 2009.

[75] Jean-Baptiste Tristan and Xavier Leroy. A simple, verified validator for software pipelining. In *POPL*, 2010.

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

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

[78] Martin Vechev, Eran Yahav, and Greta Yorsh. Abstraction-guided synthesis of synchronization. In *POPL*, 2010.

[79] Jaroslav Ševčík, Viktor Vafeiadis, Francesco Zappa Nardelli, Suresh Jagannathan, and Peter Sewell. CompCertTSO: a verified compiler for relaxed-memory concurrency. In *ACM*, 2013.

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

[81] Richard Wartell, Yan Zhou, Kevin W. Hamlen, Murat Kantarcioglu, and Bhavani Thuraisingham. Differentiating code from data in x86 binaries. In *ECML PKDD*, 2011.

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

[83] Zhaomo Yang, Brian Johannesmeyer, Anders Trier Olesen, Sorin Lerner, and Kirill Levchenko. Dead store elimination (still) considered harmful. In *USENIX Sec*, 2017.

[84] 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 *IEEE S&P*, 2009.

[85] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM: A practical concolic execution engine tailored for hybrid fuzzing. In *USENIX Sec*, 2018.

[86] Bin Zeng, Gang Tan, and Úlfar Erlingsson. Strato: A retargetable framework for low-level inlined-reference monitors. In *USENIX Sec*, 2013.

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

[88] Mingwei Zhang and R. Sekar. Control flow integrity for COTS binaries. In *USENIX Sec*, 2013.

[89] Lu Zhao, Guodong Li, Bjorn De Sutter, and John Regehr. ARMor: fully verified software fault isolation. In *EMSOFT*, 2011.

APPENDIX

A. Abstract Domains

We describe the abstract domains used in our abstract analysis below:

- *Linear Memory Isolation*:
  - `LinearMemBase`—this registers/stack slot contains the start of linear memory
  - `Bounded`—a value less than 4GB
  - `Unknown`—any value
- *Stack Safety*: The only value we track for stack safety is the size of the current stack frame.
- *Global Variable Isolation*:
  - `GlobalsBase`—the start of the global variable region
  - `LinearMemBase`—the start of linear memory
  - `Unknown`—any value
- *Control Flow Safety*: Checking control flow safety requires two different abstract domains: the jump safety domain and the call safety domain.

  The jump safety domain contains 5 possible values:
  - `JumpTableBase`—the base address of the jump table
  - `Checked`—a value that has been checked to be a valid index into the indirect jump table
  - `JumpOffset`—the offset from the base of the switch table to a valid jump target
  - `JumpTarget`—a valid jump target
  - `Unknown`—any value

  And the call safety domain contains 8 possible values:
  - `MetadataBase`—the base address of the metadata where the size of the indirect function table can be found

- `TableSize`—the size of the function table
- `Checked`—a value that has been checked against the size of the function table
- `FuncTableBase`—the base address of the function table
- `PtrOffset`—a valid offset to the function table
- `FnPtr`—a correctly checked function pointer
- `DependentPtrOffset`—a partially checked offset into the function table (see Section IV-E for more details)
- `Unknown`—any value