Provably-Safe Multilingual Software Sandboxing using WebAssembly

Jay Bosamiya, Wen Shih Lim, and Bryan Parno, *Carnegie Mellon University*

https://www.usenix.org/conference/usenixsecurity22/presentation/bosamiya

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the 31st USENIX Security Symposium is sponsored by USENIX.

Provably-Safe Multilingual Software Sandboxing using WebAssembly

Jay Bosamiya, Wen Shih Lim, and Bryan Parno

Carnegie Mellon University

Abstract

Many applications, from the Web to smart contracts, need to safely execute untrusted code. We observe that WebAssembly (Wasm) is ideally positioned to support such applications, since it promises safety *and* performance, while serving as a compiler target for many high-level languages. However, Wasm's safety guarantees are only as strong as the implementation that enforces them. Hence, we explore two distinct approaches to producing provably sandboxed Wasm code. One draws on traditional formal methods to produce mathematical, machine-checked proofs of safety. The second carefully embeds Wasm semantics in safe Rust code such that the Rust compiler can emit safe executable code with good performance. Our implementation and evaluation of these two techniques indicate that leveraging Wasm gives us provably-safe multilingual sandboxing with performance comparable to standard, unsafe approaches.

1 Introduction

Lightweight, safe execution of untrusted code is valuable in many contexts, including software plugins, third-party libraries, or even client-side code run when browsing the Web. New contexts such as dynamic content delivery networks (CDNs), edge computing, and smart contracts have created additional motivation to run untrusted code that could potentially harm its execution environment. Software sandboxing (via software fault isolation [41]) is a well-studied technique, with a long and rich history [7, 17, 21, 27, 29, 30, 37, 52], to provide this crucial primitive. Nevertheless, previous efforts to deploy it in production have failed, due to technical and marketplace hurdles (Section 2.2).

On the Web, after failed attempts with Java, ActiveX, Flash, NaCl [52], and Asm.js, a new contender for fast code execution was born—WebAssembly [11]. With lightweight, safe, portable, and fast code execution as its goals, WebAssembly (or Wasm) has rapidly become a popular compilation target for client-side code execution on the Web. Designed with sandboxing in mind, it has clean, succinct, and well-defined semantics, which, along with its portability and speed, has made it appealing for use in non-Web contexts too [8, 9, 30, 32, 42]. With the standardization of the WebAssembly Systems Interface (WASI) [42], it even exposes a POSIX-like API for programs to interact with their environment in a controlled manner. This makes it an attractive compilation target for both Web and non-Web contexts, and compilers for most popular languages, such as C, C++, Rust, Java, Go, C#, PHP, Python, TypeScript, Zig, and Kotlin, now support it as a target.

As a result, an implementation of Wasm can provide strong guarantees about the safe execution of a large variety of languages on a large number of platforms, making it an attractive narrow waist for sandboxing (Section 2.3).

However, WebAssembly's sandboxing guarantees hold only at the specification level; real Wasm implementations can, and do, have bugs (Section 2.3). These bugs can completely compromise all the guarantees provided by the specification. A plausible explanation for such disastrous sandbox-compromising bugs, even in code designed with sandboxing as an explicit focus, is that the correct, let alone secure, implementation of high-performance compilers is difficult and remains an active area of research, despite decades of work.

In this paper, we review the design space (Section 3) for executing Wasm code. Within this space, we identify a crucial gap: Wasm implementations that provide *both* strong security *and* high performance. Hence, we propose and explore two techniques, with varying performance and development complexity, which guarantee safe sandboxing using provably safe compilers. Along the way, we demonstrate that one can have safety without sacrificing execution performance.

We implement the first of our techniques (Section 4) as a compiler called vWasm, which utilizes formal methods to mathematically prove that the compiled Wasm code can only interact with its host environment via an explicitly provided API, hence ruling out problems where the compiled code, say, reads/writes to prohibited host-memory locations, or jumps to prohibited host code. We accomplish this by writing a machine-checked formal proof about vWasm's implementation. In particular, the executable code produced by the implementation is formally guaranteed to stay within the confines of the sandbox provided to it. Note that this differs from traditional compiler correctness (in the vein of CompCert [24]), which guarantees that the output program matches the input. Indeed, the two properties are orthogonal, and thus we focus on provable sandboxing. To our knowledge, vWasm is the first formally verified implementation of a multi-lingual sandboxing compiler, since it can sandbox any of the many languages with an existing Wasm backend (Section 2.3). This complements earlier work [21] that verifiably sandboxed Cminor, one of CompCert's intermediate languages, for CompCert's three backends.

Our second technique (Section 5), implemented as a compiler called rWasm, takes a different approach that targets the special nature of software sandboxing. By careful optimized lifting of Wasm code to Rust, followed by compilation down to native code, it provides high-performance execution of Wasm code while guaranteeing safe sandboxing. By leveraging Rust's safety guarantees, rWasm provides safety *without requiring any explicit proofs* from the developer. Our benchmarks (Section 6.1) show that rWasm is competitive with, or on some benchmarks even beats, other Wasm runtimes, including ones optimized for performance, rather than safety. By leveraging Rust, rWasm provides the first multi-lingual, multi-platform sandboxing compiler with provably safety guarantees and competitive performance.

vWasm, implemented in F* [39], currently compiles Wasm programs into x86-64 assembly code, although the code and proofs are designed for portability. To prove its high-level theorem, we model a subset of x86-64 semantics and prove that the produced code satisfies the sandboxing statement given these semantics. rWasm, on the other hand, is implemented in Rust, and can compile code to any architecture that is supported as a target by Rust (which covers all the widely-used architectures). It also supports the ability to conveniently customize the output program, e.g., to add inline reference monitors [7]. Both tools are available as open source.[1]

Both vWasm and rWasm are competitive in performance, with the latter providing similar performance as other performance-optimized Wasm implementations on various benchmarks. The former is faster than interpreters, but slower than unsafe compilers. We compare both implementations on qualitative aspects in Section 6.2, including development/-maintenance effort and extensibility.

As with most sandboxing tools, we focus only on protecting the host environment from the sandboxed code. Hence, we do not make any claims about the impact of buggy code on itself. We also assume that the environment is not corrupt or overly permissive in the APIs exposed to sandboxed code. Finally, protections from denial of service, speculative execution, and side-channel attacks are orthogonal and out of scope.

As we discuss in Section 7, an alternative to provably emitting sandboxed code is to *validate* that code has been properly sandboxed (cf. NaCl [52], RockSalt [29], and VeriWasm [17]). However, these approaches require a custom validator for each targeted platform. NaCl and RockSalt also rely on a custom compiler toolchain for x86/x64 to make the emitted code easier to validate. Verifying code that was not so customized, e.g., with VeriWasm, is tricky to do without rejecting legitimate programs or suffering soundness issues. For example, VeriWasm missed CVE-2021-32629, a sandbox-compromising bug in Wasmtime [46] and Lucet [1], due to improper modeling of signedness in their specification [31].

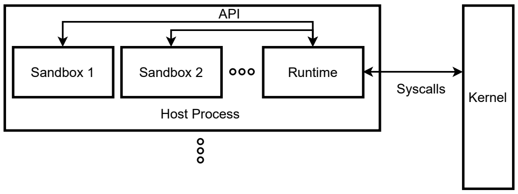

Figure 1: SFI-based intra-process sandboxing. In practice, the Host Process might be a video player, and each sandbox might contain a different codec or extension.

Of course, a specification failure is problematic both for verified compilation and for validation. However, verified compilation allows greater control over the produced code; e.g., vWasm only needs to model a small, simple fragment of x64. In contrast, validation typically handles complex assembly produced by an independent compiler. The two approaches are complementary though, and ideally both would be used.

In summary, this paper makes the following contributions.

1. An exploration of two distinct techniques to achieve provably safe, performant, multi-lingual sandboxing. We implement these as open-source tools, and evaluate them on a collection of quantitative and qualitative metrics.
2. vWasm, the first verified sandboxing compiler for Wasm, achieved via traditional machine-checked proofs.
3. rWasm, the first provably safe sandboxing compiler with competitive run-time performance. We achieve this using non-traditional repurposing of existing tools to provide provable guarantees without writing formal proofs.

2 Wasm as a Narrow Waist for Software Sandboxing

We review software-fault isolation (Section 2.1) and Wasm (Section 2.2), before discussing Wasm's unique suitability for multi-lingual, cross-platform sandboxing (Section 2.3).

2.1 Background: Software Sandboxing

As discussed in Section 1, safe, lightweight execution of untrusted code is necessary in many contexts. A popular ap-

[1] https://github.com/secure-foundations/provably-safe-sandboxing-wasm-usenix22

proach is Software Fault Isolation (SFI) [41], which limits the effects of bugs to the buggy code itself. Without requiring any special hardware, it confines any bug's impact within a user-defined boundary, typically within a module or library. Multiple such boundaries can be introduced within the same OS-level process, as shown in Figure 1. Inline checks before any potentially unsafe memory access enforce this boundary. The cost of these checks is offset by the performance savings from cheap transitions between the sandboxed code and the host process. Common techniques to implement these checks include restricting offsets through bit masks, explicitly checking bounds, or using hardware quirks like x64's zero-extend on 32-bit arithmetic. Ensuring these checks always run requires some form of Control Flow Integrity.

2.2 Background: WebAssembly

The Web has seen many technologies for client-side code execution, but all except JavaScript have fallen by the wayside. JavaScript's limitations as a compilation target, however, motivated the design of Wasm.

Wasm introduces a new virtual architecture built from the ground up with speed, safety, and portability in mind. Its virtual architecture provides a platform-agnostic solution to compilation and code execution on the Web. It is a stack-based architecture with well-defined semantics and a basic type system. Its semantics are entirely deterministic, except for floating-point NaNs. It does not have a garbage collector, and hence gives the developer (or compiler) more control over run-time performance. Despite being a virtual architecture, it is designed to be close to modern hardware, making reasoning about its execution much simpler. These properties make it a great compilation target.

In more detail, WebAssembly programs are composed of separate modules, each of which consists of collections of code, data, and associated connections to the environment (or other modules). Code lives in simply-typed functions that can access the module's memory and global variables. Memory is a (potentially growable) sequence of bytes, called linear memory, whose length is a multiple of 64 KiB. This memory is disjoint from all other parts of the module. Globals consist of named scalar values (i.e., no arrays). Functions can be called either directly, or indirectly by picking an offset in an indirect call table. Control flow within a function consists of conditionals, blocks, loops, direct jumps (conditional and unconditional), and indirect jumps. Wasm guarantees that all control flow is structured by only allowing jumps to labels of blocks (or loops) that enclose the jump. Indirect jumps are performed, similar to indirect calls, by picking an offset in an indirect branch table. All imports to, and exports from, the module are made explicit, avoiding implicit access to the environment.

2.3 Motivation: A Narrow Waist

WebAssembly's careful design enables sandboxed execution of high-performance code on the Web. However, this same design can also benefit non-Web applications, since the Wasm standard explicitly separates the core Wasm language from the specific API provided to each Wasm module by the runtime or other modules. For example, instead of offering a Web-oriented API, (say) for manipulating the DOM, many runtimes [1, 43–46, 48, 49] offer the WebAssembly System Interface (WASI) [42] API to run Wasm beyond the Web. Our compilers vWasm and rWasm are agnostic to the particular runtime API picked by the host.

Given its popularity, and the large number of compilers that support compilation *to* Wasm[2], from languages including C, C++, Rust, Java, Go, C#, PHP, Python, TypeScript, Zig, Kotlin, and more, a single compiler *from* Wasm is sufficient to immediately support sandboxed code execution for all those languages. This makes Wasm an attractive narrow waist to provide high-performance lightweight sandboxing.

Such a compiler from Wasm to (say) x86-64 is simpler in design than x64-to-x64 SFI rewriting or sandboxing. Wasm's stack-based architecture, type system, and well defined semantics all make Wasm easier to reason about than x64. It also has a drastically smaller architecture, with under 200 instructions, compared to the $\sim 1500-6000$ instructions in x64 [13, 15]. Additionally, it has no unexpected, or hardware/platform-specific behavior. While it *does* have minor non-determinism, in the form of computations involving floating-point NaNs, this is the only source of non-determinism, and thus its behavior is much easier to reason about.

As a narrow waist, then, Wasm seems to immediately provide high-performance lightweight sandboxing across all platforms, but note that the actual implementation of the compiler *from* Wasm is a critical part of the TCB for that guarantee. In particular, any bug in the compiler could threaten the sandboxing protections, and indeed such bugs have been found in existing runtimes, and would lead to arbitrary code execution by an adversary. For example, using carefully crafted Wasm modules, an attacker could achieve a memory-out-of-bounds read in Safari/WebKit using a logic bug (CVE-2018-4222), memory corruption in Chrome/V8 using an integer overflow bug (CVE-2018-6092), arbitrary memory read in Chrome/V8 using a parsing bug (CVE-2017-5088), arbitrary code execution in Safari/WebKit using an integer overflow bug (CVE-2021-30734); a sandbox escape in both Lucet and Wasmtime using an optimization bug (CVE-2021-32629); and many others. Recall that writing a high-performance compiler is already hard, and compilers from Wasm need to protect even against adversarial inputs, which makes it even harder. Indeed, there appears to be a tension between functionality (especially performance) and safety. We explore this in the next section.

[2] Any library or POSIX-compliant program (except for multi-threading and `longjmp`, which Wasm currently lacks) can be compiled to Wasm; our compilers support all current Wasm constructs.

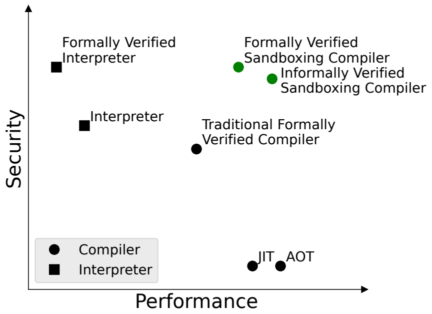

Figure 2: The Design Space for a Wasm-based Sandbox

3 The Design Space for Implementing a Wasm-based Sandbox

We consider the design space for a Wasm-based sandbox along two major axes– Security and Performance. Figure 2 shows an informal representation of the space, and the location of types of Wasm runtimes within it.

Security and Threat Model To describe the security of various Wasm-based sandboxes, we must first understand the threat model. For the purposes of software sandboxing, this is largely standard, but we describe it in particular for sandboxing Wasm modules. At a high level, the attacker has some control over execution of the sandboxed module (through buggy, or even malicious, attacker-provided module code) and wishes to "break out" into the environment outside the sandbox to obtain more control. Under our threat model, the attacker may start executing code from any of the explicitly exposed "starting points" within the Wasm module, can manipulate Wasm's linear memory at arbitrary points between Wasm executions, and can pass arbitrary arguments to Wasm functions when calling them. The attacker, however, does not arbitrarily control the environment, since if they already have that level of control, then they need not "break out" of the sandbox. Additionally, side-channel attacks, speculative execution attacks, hardware attacks (such as Row Hammer [20] or power glitching [19]), denial of service, or excessive resource usage are out of scope. Confused deputy attacks [12], where the more privileged host process is tricked into misusing its authority due to a badly designed API, are also out of scope but addressed in other orthogonal work [30].

Performance Obviously, for critical systems, security is vital. However, for deployment in production, high performance is also important. Different workloads may have different performance requirements. Short workloads that run only once have different requirements from long running ones, or ones that require multiple runs. To simplify the discussion, within Figure 2, we focus on workloads that are either long running or require many runs. Our rough categorization here is supported by quantitative measurements in Section 6.

Design Space The top-left corner of Figure 2 consists of high-safety but low-performance implementations. Interpreters occupy this space, since (barring a language bug in the implementation of the interpreter), an attacker cannot escape the interpreter. However, this safety comes at the cost of run-time performance. Verified interpreters, such as Watt's verified Wasm interpreter [47], may provide better/additional safety properties relative to unverified interpreters.

The bottom-right corner consists of high-performance but low-safety implementations. Occupied by compilers, this space is used in many production scenarios, such as browsers. Since compilers are complex software, they are prone to bugs and thus can compromise safety (see Section 2.3). Amongst the two types of compilers, ahead-of-time (AOT) and just-in-time (JIT), the AOT compilers typically produce faster execution at run time since they can afford to spend more time on optimization. In contrast, JIT compilers tend to optimize the total execution time, including compilation time, and thus spend less time optimizing.

In the middle, we have traditional formally verified compilers [22, 24]. For most languages, traditional compiler correctness is orthogonal to sandboxing safety. The former reasons about the semantic equivalence of safe input code and its output and makes no guarantees about its compilation results when given an unsafe input program, e.g., a C program with a buffer overflow. Sandboxing safety reasons about the output code independent of the input program (or its safety).

However, Wasm is special since its semantics are (almost) deterministic. This means that *if* one formally proves this determinism, and composes that with traditional compiler correctness, then sandboxing safety could be proven from it. Thus, a traditional formally verified compiler, while safer than unverified compilers, is still imperfectly safe.

Finally, we have the upper-right quadrant of Figure 2, which shows our goal, where neither safety nor performance are compromised. We achieve this goal using the traditional means of formal verification for vWasm, while using a non-traditional repurposing of existing tools for rWasm. We defer the discussion of our techniques to Sections 4, 5 and 6.

In Section 6, we consider additional axes, including compilation time, development complexity, portability, and extensibility.

4 vWasm: A Formally Verified Sandboxing Compiler

Inspired by previous successes in constructing formally verified compilers for C [24] and ML [22], we construct vWasm to compile Wasm to provably sandboxed code. Previous work primarily focused on proving that correct input code is faithfully compiled to correct output code, whereas vWasm focuses on proving that all input code (regardless of correctness or even malice) is compiled to safely sandboxed code.

Concretely, we implement vWasm in the F* proof assistant (Section 4.1), following a relatively standard compilation pipeline (Section 4.2). We formally state and prove vWasm's guarantee that all output code will be properly sandboxed (Section 4.3), relative to a mechanized semantics of a subset of the x64 ISA. Finally, we summarize some lessons learned from vWasm's development (Section 4.4).

4.1 Background: Formal Verification and F*

Formal verification of software consists of writing a formal statement of the property we wish to prove about the software, and then writing a formal proof that shows this statement is true. The proof is machine checked and thus provides the highest degree of assurance in its correctness. In contrast to techniques such as software testing, fuzzing, and manual reviews, formal verification is able to reason about *all* execution paths, provided *any* input. This means that behaviors like buffer overflows, use-after-frees, etc. are completely ruled out. We describe vWasm's top-level property, as well as our proof strategy, in Section 4.3.

Our verification tool, F* [39], is a general-purpose functional programming language with effects, built for formal verification. Syntactically, it is closest to languages from the ML family (such as OCaml, F#, or SML). It has the full expressive power of dependent types, and has proof automation backed by Z3 [4], an SMT solver. Code written in F* can be extracted to multiple languages, and for vWasm, we use F*'s OCaml extraction. Proofs are written within vWasm as a combination of pre-/post-conditions, extrinsic lemmas, and intrinsic dependently-typed values. Also, to aid in proof management, we regularly use F*'s layered effects [34].

4.2 Compilation Strategy

vWasm is implemented as a compiler from Wasm to x86-64 (abbreviated as x64 henceforth), but it is designed to keep most of its code and proofs generic with respect to the target architecture. Here, we describe the process of compiling to x64, but the techniques generalize in a straightforward way to other architectures such as ARM. In compiling from Wasm to x64, there are three important conceptual stages: (i) a frontend which compiles Wasm to an architecture-parametric IR, (ii) a sandboxing pass which acts upon the architecture-parametric IR, and (iii) a printer which outputs the x64 assembly code.

The frontend for the compiler is both untrusted and unverified. This means that one neither needs to trust its correctness for the overall theorem statement to be true, nor does one need to write proofs about it. Note that this is in stark contrast with traditional compiler verification, where any stage of the compilation must either be trusted or verified. This means that we are free to use any compiler technology for the compiler's frontend, including arbitrarily complicated optimizations, as long as it outputs code within our architecture-parametric IR. This drastically reduces the development cost of such a compiler, compared to a traditional verified compiler, and it can also allow for fast code by allowing full usage of compiler optimization research. Since compiler optimization is orthogonal to our primary goal, for vWasm's frontend, we implemented only a simple register allocator and a basic peephole optimizer. We leave such optimizations for future work.

On the other end of the compilation pipeline is the x64 assembly printer, which is trusted to be correct. We discuss vWasm's overall TCB in Section 4.3, but we note that the printer is largely a straightforward one-to-one translation of our IR to strings, making it fairly simple to audit.

Finally, the sandboxing pass, which lies between the above two, is untrusted but verified to be correct. We define this formally in the next subsection, but informally, this means that the sandboxing code has been proven (and the proof mechanically checked) to produce safely sandboxed code, given any input. Within the sandboxing pass, all accesses (reads or writes) into the Wasm module's linear memory, indirect function call table, imports, globals, etc. are proven (sometimes after suitable transformations) to be safe.

In particular, to perform safe sandboxing, we bound accesses primarily via a bitwise-AND operation whenever possible, falling back to a check-and-branch-based bound otherwise. Admittedly, this runs counter to the WebAssembly specification, which dictates that *any* access outside linear memory must immediately trap. However, we allow for this small difference in semantics to support simplified sandboxing. For valid program executions, this does not impact their execution trace. However, upon an invalid out-of-bounds access, instead of trapping and exiting, the module may continue execution while corrupting its own memory space.[3] This transforms a security bug into a correctness (but sandbox-safe) bug. In some scenarios, this may be undesirable, so vWasm could be modified to instantly terminate.

To prove sandbox safety, we additionally prove that the sandboxing pass also guarantees (a restricted form of) Control-Flow Integrity (CFI) that ensures that any checks performed for sandboxing cannot be bypassed, and thus must be obeyed. Due to the convenient *explicit* split between different types of accesses in Wasm (e.g., linear memory is disjoint from the indirect call table and globals), sandbox checks can be safely elided in most cases except for direct linear memory accesses. This elision too is proven safe against the x64 machine model.

The sandbox used in vWasm has a fixed compile-time size, but since many Wasm programs need the ability to grow memory, we emulate the size accessible to the program, while using the sandbox size as a constant upper bound[4]. The loca-

[3] Despite allowing such corruption, vWasm does *not* depend on external assumptions like W^X to prevent code-modification attacks. Explicit checks force writes to stay within explicitly-provided data-only regions. Our proofs demonstrate these checks suffice and cannot be bypassed.

[4] Wasm uses 32-bit addressing; thus Wasm programs cannot refer to memory beyond 4GiB. vWasm supports setting the limit to this maximum, so any limitations due to memory-size bounds are due to Wasm itself. Wasm's

```
type operandi =
  | OConst : n:int → operandi
  | OReg : r:regi → size:rsize → operandi
  | OMemRel : offset:maddr → operandi
  | ...

type ocmp =
  | OEq32 : o1:operandi → o2:operandi → ocmp
  | ...

type ins_t =
  | Add32 : dst:operandi → o2:operandi → ins_t
  | ...

type precode =
  | Ins : i:ins_t → next:loc → precode
  | Cmp : cond:ocmp → t:loc → f:loc → precode
  | Call : tgt:target_t → onreturn:loc → precode
  | ...

type code = list precode
type ok_t = | AllOk | MemFailure | AstFailure | ...

type state = {
  ok:ok_t; ip:loc; reg_i:int^→nat64;
  mem:heap; stack:stack; ...
}

let eval_step (c:code) (s:state): state = ...
```

Figure 3: Sample of our machine model for x64 in F*

```
val sandbox_compile
  (a:aux) (c:code) (s:erased state): Err code
    (requires (
        (s.ok = AllOk) ∧
        (reasonable_size a.sb_size s.mem) ∧
        (s.ip `in_code` c) ∧ ...))
    (ensures (fun c' →
        forall n. (eval_steps n c' s).ok = AllOk))
```

Figure 4: Simplified theorem statement in F* for provably safe sandboxing in vWasm.

tion of the sandbox in memory itself is not fixed at compile time, but instead is chosen at run time, allowing for extra mitigations, such as Address Space Layout Randomization.

4.3 Provably Safe Sandboxing

Reasoning about sandboxing code involves first defining a machine model, and then defining what sandbox safety is within this model. Our machine model covers the subset of x64 targeted by the compiler. A simplified version of this model, written in F*, is in Figure 3. The complete model can be found in our open-sourced code. Note that these semantics for x64 are defined as small-step semantics, allowing for reasoning about even potentially infinitely running code. Within the definition of the x64 `state`, the `ok` field is crucial for defining safe sandboxing. This field is set to the value `AllOk` if and only if, until that point in execution, nothing invalid has occurred. Crucially, this also means that no accesses outside the memory allocated to the module have occurred.

Sandboxing is safe if and only if, informally, starting from any initial `AllOk` state, executing the sandboxed code for any number of steps leads to an `AllOk` state. Figure 4 shows the overall statement of this theorem for the sandboxing pass, more formally written in F*. This statement, written as pre- and post-conditions for the sandboxing pass `sandbox_compile`, shows that any code output by the sandboxer is formally guaranteed via the machine-checked proof to be safe. The pass takes two arguments `a` (auxiliary data) and `c` (the input program), and a computationally-irrelevant argument `s` (the initial state of the program, which is used for reasoning in our proofs, but that is erased when running the compiler), and returns output code `c'` under the custom effect `Err` (which allows the compiler to quit early upon error, for example if it finds a call to a non-existent function). The statement guarantees that as long as the pre-conditions in the `requires` clause are satisfied, the post-condition in the `ensures` clause provably holds on the produced output code. The conditions say that the initial state must be safe, have a reasonable sandbox size, and start from a valid location in the code; if these conditions are met, the output code `c'` will be safe when executed for any number of steps `n`.

The safety of the code means that it cannot access memory outside the allowed boundary. In particular, this means that any invalid (or even malicious) code provided to the sandboxer is made safe, and thus issues such as buffer overflows can only corrupt the module's internal state and nothing outside it. This property is distinct from traditional compiler correctness, which is predicated upon the input code being safe.

To better understand this guarantee, we must understand the Trusted Computing Base (TCB) under which it holds, which then explains potential limitations. Since this proof is written in F*, our TCB includes all that comes with it, specifically, F*, Z3 (its SMT backend), and OCaml (which is what we extract the F* code to, in order to execute). Additionally, our TCB includes the x64 machine model, the full theorem statement from Figure 4, and the trusted x64 assembly printer, as well as the assembler that converts the printed assembly into machine code. Note that all of these portions of the TCB would be required even for the implementation of a traditional verified compiler, but crucially, we do not have the semantics of the input language in our TCB, while traditional compiler correctness would necessarily need it in order to state its semantic equivalence (or simulation) theorem.

Our proof strategy consists of multiple parts. First, we use small-step semantics, which allow for more fine-grained reasoning about code execution. Next, we combine some instructions into groups of instructions. A group of instructions can be any positive number of instructions executed in a straight-line fashion. We then show a CFI property that this

real-world usage suggests this is not very limiting in practice.

grouping cannot be escaped from; i.e., a group can only be entered at its start, and only exited at its end, thereby behaving similar to a (subset of a) basic block. For normal control flow (such as straight-line code, or for conditional/unconditional direct jumps) this is straightforward, but more effort is required for indirect control flow (indirect function calls, indirect jumps, lookup/jump tables, and function returns). Next, we show that each group maintains the `AllOk` invariant; i.e., no invalid memory accesses or any other transition to an invalid state has occurred yet. By ensuring that all potentially unsafe accesses are correctly checked within a group, this invariant follows as a result. Finally, we prove that this invariant suffices to show that executing any number of steps in the program maintains `AllOk`.

4.4 Useful Insights

In the process of implementing and verifying vWasm, we uncovered some useful insights, which are likely to generalize to any formally verified sandboxing compiler. The first of these is that proving sandboxing is far more convenient with small-step semantics than it is to do it with big-step semantics along with a full execution trace. We initially started with the latter, but found that the proofs were convoluted and were simplified significantly by switching to the former. This is because small-step semantics allowed us to more naturally state and work with the invariant along groups of instructions. At its core, each group of instructions behaved like a single "medium step" formed from a finite collection of small steps, and this property is more naturally expressed as combining small steps on an unchanged program, rather than as a big step on the restricted program consisting of only the group.

Next, we found, to our surprise, that an unstructured IR was more convenient to work with to prove sandboxing. Wasm is carefully designed as a structured virtual architecture, not allowing for arbitrary `goto`s, and such structure generally simplifies proofs. However, Wasm introduces multiple distinct control-flow constructs consisting of loops, blocks, conditionals, branches, conditional branches, indirect branches, and direct/indirect calls. Each of these contribute non-trivial complexity to the proof, and hence we found that making the IR unstructured quite early in the compiler simplifies the overall implementation and associated proofs. In particular, vWasm quickly moves to an IR based on an unstructured Control Flow Graph (CFG) and maintains this right down to emitting x64 assembly at the printer. This means that all the control flow constructs are unified and do not require special handling.

Another useful property we noticed is that most of the proofs for the sandboxing are architecture independent. While we have implemented our compiler to only emit x64, very few proofs depend on x64-specific behavior. Instead, most of them rely more abstractly upon whether an instruction may access a certain region of memory or not. Thus, we believe that the compiler and its proofs are *almost* architecture independent. Specifically, with some more proof engineering, location modeling could be made more abstract, similar to that done by Bosamiya et al. [2], at which point the proofs would be practically architecture independent. We leave this for future work.

Finally, to implement a convenient-to-reason-about CFG-based semantics, a list of instructions with jumps indexing into it is useful. However, this makes for a very slow compiler, since updates to a functional list are expensive in languages like OCaml that use immutable linked lists. In order to balance proof complexity and implementation performance, we implemented a new functional data structure which we call an Append Optimized List (AOL). An AOL contains all the operations that one might require from a functional list, and each of these operations are proven to correspond to the equivalent operation on the simulated functional list. However, these operations are optimized for performance by implementing AOL as a tree that allows for delaying operations. For example, appending two functional lists is a linear time operation (in the length of the first list), but appending two AOLs is a constant-time operation, since it only requires the creation of a single node that points to both. Other operations which are faster on AOLs than on functional lists include: `length`, `zip`, `unzip`, `repeat`, `split`, `get_at_index`, and `update_at_index`. Since these operations are proven to match the functional specification, proofs written with lists in mind work directly with no changes required; however, the overall performance of the compiler is improved by an order of magnitude (Section 6.1.2).

5 rWasm: High-Performance Informally-Proven-Safe Sandboxing

rWasm leverages the insight that Rust (Section 5.1) can provide *both* safety (Section 5.3) and good performance, if we employ a suitable compilation strategy (Section 5.2) that plays to the strengths of Rust's optimization strategies (Section 5.4). rWasm's approach also makes it surprisingly simple to instrument the produced code with reference monitors (Section 5.5).

5.1 Background: Rust

Rust [26, 40] is a systems programming language with a strong focus on performance, reliability, and safety. Developed originally for use in Firefox, it has since gained use in the industry for security- and safety-critical software components. Amongst other goals, Rust aims to eliminate memory safety errors entirely, and it does so through a memory-ownership discipline. This provides safety without garbage collection, which practically all popular memory-safe languages previously required. Rust allows a developer to write high-level code, with low-level control when needed. However, as a systems programming language, there might be certain scenarios (e.g., writing an OS) where one might need more control than directly allowed by the language, like directly accessing an arbitrary location in memory. Rust provides an explicit escape hatch for this via the keyword `unsafe`.

However, Rust guarantees that correctly typed code without `unsafe` (ensured by the declaration `#![forbid(unsafe)]`) will always be memory safe. The Rust community takes this guarantee *extremely* seriously, and considers any unsoundness in its type system to be immediately security critical (even if it may not actually be exploitable), assigns a CVE to it, and then works to fix the unsoundness [51]. Given the prevalence of Rust in industry, and how seriously the Rust team takes unsoundness bugs, safe Rust is thus battle-tested to be memory safe, even if not (yet) proven to be so. Early efforts towards formalization of Rust and its security guarantees have already begun, such as with the RustBelt [18] project and Oxide [50].

5.2 Compilation Strategy

rWasm compiles WebAssembly code to safe Rust. It consists of (i) a frontend that parses the Wasm binary into an internal intermediate representation, (ii) a stack and dead code analyzer, and (iii) a backend printer, that prints the intermediate representation into Rust code. This Rust code is then fed into the Rust compiler (`rustc`) to produce machine code.

We implement all stages of rWasm in safe Rust, and no stage of the compiler needs to be verified or trusted. This means we do not need to depend upon the safety or correctness of any part of rWasm for the safety of the produced executable machine code. Instead, the safety of the produced code simply comes from the lack of any `unsafe` in the generated Rust code. We discuss this further in Section 5.3. The compiler itself is thus free to generate code however it likes, but not all approaches produce efficient code. We discuss techniques to produce efficient code, amongst other details in Section 5.4.

5.3 Provably Safe Sandboxing

Our key insight for rWasm is that emulation of low-level code can be done by lifting it to a high-level language, which provides the guarantees of the high-level language to the low-level code under emulation. The high-level property that we want for provably safe sandboxing is essentially memory safety, meaning that the sandboxed code cannot access any memory that is not explicitly allocated to it by the runtime in the host process. Thus, compiling Wasm to *any* type-safe language (e.g., OCaml or Haskell) would provide provably-safe sandboxing through guaranteed object integrity, lack of type confusion, lack of out-of-bound memory accesses, etc. without trusting the generated type-safe code. Contrast this with `wasm2c` [43], which requires trusting the compiler, or its generated C code, since C does not guarantee memory safety.

The usual side effect of such lifting is that it leads to either unpredictable or bad performance. We recognize, however, that Rust presents us with a new opportunity to provide high-level guarantees without necessarily suffering this side effect. We have to be careful to actually eliminate it, but due to Rust's focus on performance, and lack of a garbage collector, it is feasible to obtain good and predictable performance by lifting code to Rust.

Since safe Rust's type-safety guarantees memory safety, we can thus informally prove memory safety by ensuring that there is no usage of `unsafe` in the produced code. We can achieve this simply through the use of a `#![forbid(unsafe)]` declaration. The TCB in this scenario then is only the Rust compiler. More explicitly, neither rWasm nor its generated code needs to be trusted for memory safety. While the Rust compiler itself is large, the Rust team takes type- and memory-safety extremely seriously, and thus this provides (informally) provably safe sandboxing.

Astute readers will note that sandbox safety in any type-safe language also depends on the language's runtime libraries. Fortunately, rWasm imports nothing, uses only allocation-related features (for `Vec`), and even eliminates dependency on the Rust standard library via the `#![no_std]` directive. As with any sandbox, care is required when exposing an API to sandboxed code [30] (e.g., to avoid APIs enabling sandbox bypasses directly or via confused-deputies), but such concerns are orthogonal to sandbox construction.

5.4 Useful Insights

Here, we describe an optimization-friendly collection of techniques to preserve WebAssembly semantics in Rust. Figure 5 illustrates the net effect of these techniques when compiling an example Wasm function that takes a 32-bit integer argument and computes the sum of positive integers up to that number. Note that the simple Wasm code becomes seemingly complex Rust code. However, the Rust code is written to be optimization friendly, and thus the Rust compiler is able to optimize it nearly all away, even recognizing that this convoluted looking code can be optimized via the mathematical closed form $\sum_{i=0}^{a} i = \frac{a(a+1)}{2}$. We briefly summarize key design decisions below.

The first challenge for any Wasm compiler is mapping Wasm's stack-based virtual machine to finite-register hardware. Rather than write our own register allocator for rWasm, our stack analysis pass emulates the stack-based machine via an infinite-register machine, where Rust variables are used to represent the "infinite" registers (notice in Figure 5a that the Wasm code only uses two stack slots, which correspond to Rust variables `slot0` and `slot1` in Figure 5b). We represent Wasm's non-stack locals and function arguments as scalar-typed Rust variables (see `local0` and `local1` in Figure 5b). This allows us to piggy-back on `rustc`'s excellent register allocation routines to obtain good performance.

To further simplify rWasm's implementation, we observe that Rust is adept at eliminating the repeated wrapping and

```
(func (param i32) (result i32)
  (local i32)
  loop (result i32) ;; []
    local.get 0       ;; [a]
    i32.const 1       ;; [a, 1]
    i32.lt_s          ;; [a<1?]
    if (result i32)   ;; []
      local.get 1     ;; [s], return
    else
      local.get 0     ;; [a]
      local.get 1     ;; [a, s]
      i32.add         ;; [a+s]
      local.set 1     ;; [], s ← a + s
      local.get 0     ;; [a]
      i32.const 1     ;; [a, 1]
      i32.sub         ;; [a−1]
      local.set 0     ;; [], a ← a − 1
      br 1            ;; continue
    end
  end)
```

(a) WebAssembly code. The status of the stack after each instruction's execution is shown as comments, where the stack grows towards the right.

```
fn func_0(&mut self, a: i32) → Option<i32> {
  let mut local0 = a; let mut local1 = 0i32;
  let mut slot0: TV;  let mut slot1: TV;
  'lbl0: loop {
    slot0 = tv(local0);
    slot1 = tv(1i32);
    slot0 = tv((slot0.vi32()?) <
             (slot1.vi32()?));
    'lbl1: loop {
      if slot0.vi32()? != 0 {
        slot0 = tv(local1);
      } else {
        slot0 = tv(local0);
        slot1 = tv(local1);
        slot0 = tv(slot0.vi32()? +
                   slot1.vi32()?);
        local1 = slot0.vi32()?;
        slot0 = tv(local0);
        slot1 = tv(1i32);
        slot0 = tv(slot0.vi32()? -
                   slot1.vi32()?);
        local0 = slot0.vi32()?;
        continue 'lbl0;
      } break;
    } break;
  }
  Some(slot0.vi32()?)
}
```

(b) Simplified Rust Output from rWasm

```
func_0:
        test    esi, esi
        jle     .A
        lea     eax, [rsi - 1]
        lea     ecx, [rsi - 2]
        imul    rcx, rax
        mov     edx, eax
        imul    edx, eax
        shr     rcx
        add     edx, esi
        sub     edx, ecx
        mov     eax, 1
        ret
.A:
        xor     edx, edx
        mov     eax, 1
        ret
```

(c) Compiled to x64

Figure 5: Compilation via rWasm of a program to find the sum of the first a positive integers.

unwrapping of tagged unions (i.e., sum types) without any performance penalty. Hence, we implement a custom tagged `enum` in Rust that can hold any of the native Wasm types. As shown in Figure 5b, `tv` wraps and tags values of any scalar type into the `TV` tagged `enum`; the original values can then be extracted back to a scalar type, say `i32`, while checking the tag using `vi32()?`. Using a tagged `enum` means that all Rust variables have the same type, so rWasm's stack analysis only needs to track the overall stack size, not the types of the values on the stack at any given moment during program execution. For example, given the Wasm code, `i32.const 5, drop, f64.const 3.14`, rWasm can reuse the same Rust variable, even though the stack slot has different types during execution. Furthermore, this approach makes the polymorphic Wasm instructions `select` (which picks one of the elements of the stack based upon the top element) and `drop` (which drops the top element of the stack) trivial to implement.

To handle WebAssembly's wide range of control flow constructs, we need to compile them down to those supported by Rust. While Rust does not support unstructured `goto`s, it does have the ability to `break`/`continue` to any outer loop (the labels `'lbl0` and `'lbl1` in Figure 5b correspond to the respective block structure in Figure 5a). We use this, along with conditionals and `match` expressions to implement and emulate all the intra-function control flow constructs in Wasm. Direct function calls translate trivially to direct calls in Rust, but indirect function calls have multiple design choices, such as inlining a dispatch routine, or having multiple type-disjoint dispatch routines, or calling out to a single common dispatch routine (requiring serialization of arguments on the stack and a type check and deserialization at the dispatch routine). In practice, we found the single dispatch routine to work best, for both compile-time and run-time performance.

For WebAssembly's linear memory, there are multiple design choices both for how to implement it, as well as how to access it. The first decision is whether to implement it as an overcommitted allocation with memory-size emulation (similar to that in vWasm) or to simply utilize a heap-allocated resizable array of bytes (i.e., `Vec<u8>`). The second decision is a choice between check-and-panic vs wrapping memory access. Each of these choices involves a trade-off between compile-time and run-time performance. Based on our microbenchmarks (Section 6.1), we chose check-and-panic with a resizable `Vec<u8>` as our default. While this introduces explicit checks at each access, `rustc` optimizes many of them by eliminating repeated or unnecessary checks statically.

Wasm's mutable global variables might initially seem difficult to implement in Rust, since Rust requires `unsafe` to read or write mutable globals (or `static mut` in Rust parlance). However, this has a simple fix familiar to most functional programmers: the state monad. In fact, rWasm even handles Wasm's linear memory this way. In particular, we represent the emulated Wasm module as a Rust `struct` whose associated methods emulate the Wasm module's functions. This means that the module's state is passed into each function (via `&mut self`), and the globals and linear memory can be stored safely within the Rust `struct`.

Finally, rWasm models integer overflow semantics to explicitly match WebAssembly semantics. By default in Rust, integer overflows cause panics in debug builds, and wrap in release builds. Since Wasm's integer overflow semantics are to always wrap, we explicitly perform wrapping arithmetic in Rust. This ensures that we always match WebAssembly semantics, even in debug builds. We omit this in Figure 5b due to space constraints.

5.5 Extensions

Implementing a low-level (virtual) architecture emulator via lifting to a high-level language, as we do in rWasm, comes with some extra benefits. One such benefit is that it is easy to build code tracers and Inline Reference Monitors (IRMs) in the spirit of SASI [7] (which describes how to instrument Java and x86 byte code to enforce security policies expressed as security automata). Another benefit is that the Rust compiler is able to jointly optimize the IRM and Wasm module's code, since both are part of the same generated Rust source. In fact, one could even consider the sandboxing access checks to be a special case of such IRMs. Within rWasm, we currently have multiple tracers that can be enabled if the user chooses, including function-level tracing, instruction-level tracing, and memory-access tracing, taking 75, 10, and 70 lines of code respectively to implement. Anecdotally, we found it easy to debug rWasm and its output during development due to IRMs. It would not be difficult to extend rWasm with other IRMs, even with very high precision, such as byte-level granularity run-time taint analyzers. Such extensions could potentially allow one to implement various sanitizers, such as Address-Sanitizer (ASan) [38], without introducing much overhead, or indeed even needing source (which compiling with ASan requires). We leave such extensions for future work.

6 Evaluation

We evaluate vWasm and rWasm against multiple popular Wasm runtimes. These include interpreters (wasm3 [44] and WAMR [49] in interpreter mode), JIT compilers (Wasmer [45] and wasmtime [46]), and AOT compilers (wasm2c [43], WAMR [49] in AOT compilation mode, WAVM [48]). We choose these runtimes for comparison as they are both popular, as per GitHub stars, and also support the WebAssembly System Interface (WASI) [42], allowing for direct comparisons. We also investigated Lucet [1], which uses Wasm for sandboxing, although without any guarantees.[5] Unfortunately, despite extensive efforts with multiple versions, we were unable to get Lucet to execute any of our 30 benchmarks, and thus we exclude it from comparison.

We evaluate these Wasm runtimes on both quantitative (Section 6.1) and qualitative (Section 6.2) metrics. All benchmarks run on a system with an AMD Ryzen 3700x processor and 64 GB of memory. We compile benchmarks to WASI-compliant Wasm binaries using Clang [3] with the `-O3` optimization level. In addition, for comparison against native (non-sandboxed) execution, we compile directly to native x64 code also using Clang with `-O3`.

6.1 Quantitative Benchmarks

6.1.1 Execution Time

As discussed in Section 3, run-time performance is critical for practical adoption in most applications. Hence, we measure execution time for our compilers and our various baselines using the PolyBench-C benchmark [33] suite, consisting of thirty programs, which has been a standard benchmark suite for Wasm since its inception [11].

Figure 6 summarizes our results, showing the normalized execution time of the benchmarks on the Wasm runtimes. Each point in the chart is the ratio of the mean time taken to execute the benchmark with the particular runtime vs. the mean time taken to execute by compiling the C code directly to non-sandboxed x64. We run each benchmark between 10 and 1000 times, based upon the time taken to run the particular benchmark. The mean of the different normalized execution time, and the 25% and 75% quartiles are shown for each runtime. Appendix A shows a detailed breakdown and further analysis.

The results indicate that, unsurprisingly, compilation strictly dominates interpretation for run-time performance. Note that, as seen in the original Wasm paper [11], we too find some benchmarks execute *faster* when compiled via Wasm than when compiled directly to native code.

With respect to our compilers, we see that rWasm is competitive even with the compilers which are optimized for speed, and not necessarily safety, only slower by 3% to 26% on average than the first three of the four faster runtimes on the list (wasm2c, WAMR in AOT compilation mode, and wasmtime respectively). The fastest, WAVM, is almost twice as fast as rWasm on average, but on some of the longer running PolyBench-C benchmarks (such as `2mm`, `3mm`, and `gemm`), rWasm is more than twice as fast than WAVM, and thus we see that relative performance can vary drastically based upon workload. vWasm consistently outperforms the interpreters on all benchmarks (by 30% on benchmarks like `reg_detect` and `fdtd-apml` to 600% on benchmarks like `cholesky` and `ludcmp`). However, while on average it is 2× to 3× faster than the interpreters, it is slower than the other compilers by 3.5× to 7.5×. However, Figure 6 also shows the execution time for vWasm with the sandboxing pass disabled. We find that the run time is marginally affected (by only 0.2%). This indicates that almost all of the slowdown, compared to other compilers, is due to the unverified portion of the compiler, which can be improved without needing to write any new proofs or even impacting existing proofs. In particular, replacing the simple register allocator and introducing stan-

[5] Indeed, one of the developers remarked, “We're just constantly fixing bugs with it” [28].

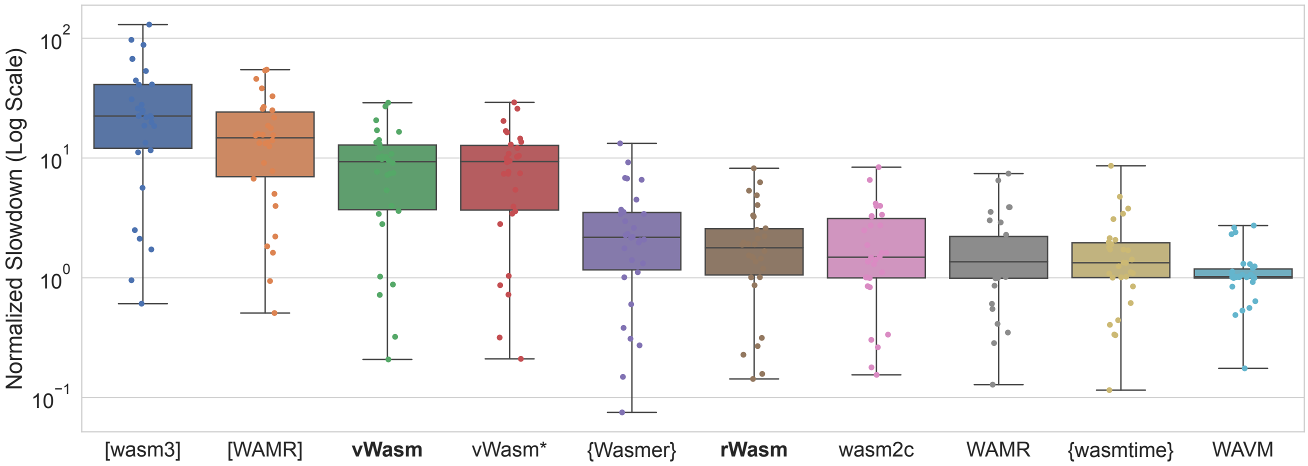

Figure 6: Mean execution time of PolyBench-C benchmarks across the Wasm runtimes, normalized to pure native execution. Interpreters have square brackets; JIT compilers have braces; the rest are AOT compilers. vWasm* disables sandboxing.

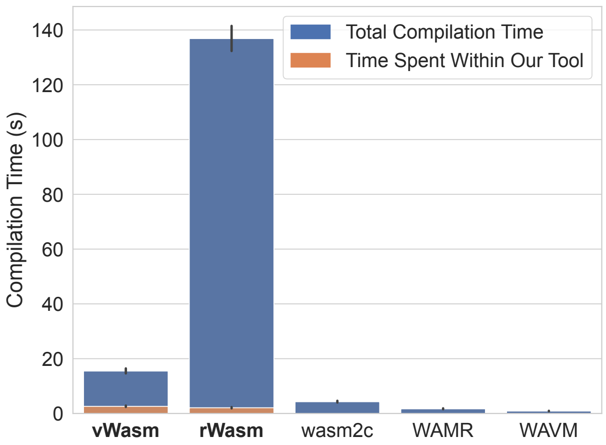

Figure 7: Mean compilation time for the PolyBench-C benchmarks across the Wasm AOT compilers.

dard optimizations (e.g., common subexpression elimination) should improve the performance of code compiled by vWasm significantly, without any proof effort.

Microbenchmark: Sandboxing Memory Accesses There are multiple design choices for how memory accesses are implemented (Section 5.4). We compare these in Figure 8, both on our regular AMD-based test bench, and on a system with an Intel i9-9900K with 128GB of RAM. These violin plots show the time taken to read a single 64-bit integer from memory, using different methods to confirm the read's memory safety. We represent the three approaches, namely No Sandbox (N), Check-and-Bound (C), and Wrapping with Bitwise AND (B), as rows/violins in the plot. N is shown only as an unsafe

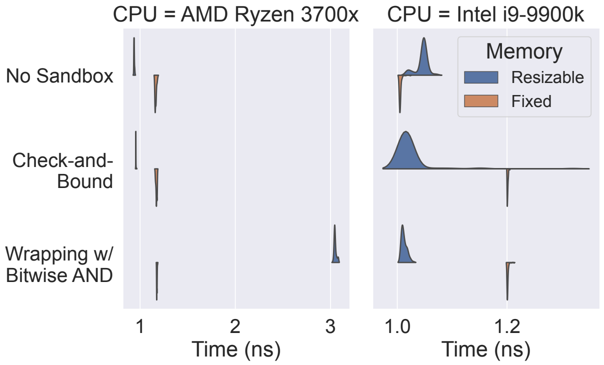

Figure 8: Design choices for sandboxing memory accesses.

baseline to compare the safely sandboxing C and B. In each violin, the top half shows the probability density of time taken (collected from $\sim 2 \times 10^9$ samples for each configuration) to perform a single read from resizable memory (via `Vec<u8>`), while the bottom half shows the same from a fixed-size array. On both CPUs, for fixed-sized arrays, the difference between C and B is negligible. However, on resizable memory, we find that C is three times faster than B on the AMD CPU, while on the Intel, B is only marginally faster than C. Thus, we find a significant difference in picking the better approach on modern Intel and AMD CPUs. More surprisingly, C either almost meets, or significantly beats the performance B, contrary to conventional SFI wisdom.

6.1.2 Compilation Time

For some scenarios (e.g., Web apps), compilation time matters, since it sits on the critical path for an impatient user.

vWasm Component	Lines of Code	Verif. Time (s)
x64 Semantics	2068	114
Printer	1458	45
Parser	558	19
Frontend	2747	57
Register Allocator	1822	87
Optimizer	185	8
Sandboxing	3607	450
AOL	737	9
Layered Effects	443	8
Misc	1160	22

rWasm Component	Lines of Code
AST + IR	384
Parser	888
Stack Analysis + Printer	2157
IRMs	155
Misc	109

Table 1: Development Effort. The first two components of vWasm are its primary TCB.

For other scenarios (e.g., installing dynamic client code at a CDN), compilation time happens off the critical path.

For completeness, Figure 7 shows the mean compilation time needed for each of the Wasm compilers. For our compilers, we also show the split between time spent within our implementation, and time spent in the tool that runs after.

For vWasm, approximately 17% of the 16 seconds is spent within the compiler, and the rest is spent in the assembler. For rWasm, 1.5% of the 137 seconds is spent within our compiler, and the rest is spent within the Rust compiler, which is known for slow performance. We note however, that any improvements in the `rustc`'s compilation time will automatically improve our overall compilation time without requiring any changes to rWasm.

Faster Compilation with Append Optimized Lists As discussed in Section 4.4, we introduce an efficient implementation for functional lists with operations proven functionally equivalent to standard functional lists. When measured on the compilation times for PolyBench-C, they reduce compile times from 25+ seconds to 2.5s, which is an order of magnitude improvement. The impact on verification time and effort (apart from proving the AOL functionally correct) is negligible, since AOLs provably meet the specification for standard functional lists, meaning that they behave as a "free" drop-in replacement.

6.1.3 Development Effort

Next, we quantify the development effort needed to implement both vWasm and rWasm. The former took approximately two person-years to develop, including both code and proofs, while the latter took one person-month. This stark contrast is a testament to the daunting amount of work formal verification requires, even with modern, automated tools like F*. It also illustrates the significant benefit of rWasm's carefully leveraging Rust's investment in safety. We describe the development effort qualitatively in Section 6.2.

Property	vWasm	rWasm
Safety Guarantee	Theoretically stronger	Provable w/ non-standard TCB
Initial Impl.	Less efficient	More efficient
Maintenance	Less efficient	More efficient
Static Properties	More extensible	Less extensible
Run-Time IRMs	Less extensible	More extensible

Table 2: Summary of the Qualitative Comparisons

As another quantitative measure, we include lines of code for both tools in Table 1, split by high-level components, along with total time taken for F* to verify the components of vWasm. We note however, that this only shows overall time taken to verify; it is not indicative of the interactive verification cycle, which is what comprises the majority of vWasm's development time. To keep the interactive proof engineering cycle tolerable, most proofs in our code base take under ten seconds to verify, and even the most time consuming proofs are checked in under two minutes.

6.2 Qualitative Evaluation

Here, we qualitatively compare vWasm and rWasm against one another, as two important points in the design space (Section 3). We summarize the comparison in Table 2.

Safety The level of assurance provided by both vWasm and rWasm is extremely high, since both provide provable safety. However, only the former is formally verified and reasons directly about the generated assembly code. Thus, one might argue that it is theoretically safer. From a more practical view, we need to consider their respective TCBs to understand safety. The TCB for vWasm is standard in many verification papers, in that it includes the verification tool (and its dependencies) as well as the model we are verifying against (here, the x64 machine model). For rWasm, the TCB is non-standard since it includes trusting the compiler of a language that is only a decade old, but is usually not part of verification. However, Rust is committed to memory safety, and is trusted for many security-critical applications in the industry, and thus may be considered safe enough. The decision for which to pick, purely based upon safety, thus relies on which TCB one considers more trustworthy, since the two are practically disjoint, and thus not directly comparable.

As a sanity check, we also confirm that exploit attempts are caught. Specifically, we implement an end-to-end image conversion scenario using netpbm 10.26 (vulnerable to CVE-2008-0554) and libjpeg-turbo 2.1.1. We compile them to separate Wasm modules, and use them to convert GIFs to JPEGs via a trusted driver program that hosts them as separately sandboxed libraries in the same process. On an example input, the rWasm-compiled and vWasm-compiled versions show a mean slowdown of 1.301× and 3.825× across 100 executions respectively, compared to the equivalent native program without Wasm-based compilation and sandboxing. We also test the proof-of-concept for CVE-2008-0554 on all three versions. This causes the native version to a crash with attacker-controlled state (potentially leading to arbitrary code execution). In contrast, unsurprisingly, both vWasm- and rWasm-compiled versions are able to successfully detect the buffer overflow and terminate the module's execution, returning back safely into the driver program.

Development Effort: Initial Implementation As noted in Section 6.1, vWasm took much longer to implement than rWasm. The reasons for this are many fold: (i) developing verified software continues to be significantly more difficult than unverified software; (ii) a full compiler down to assembly is more complicated than one to a high-level language, due to low-level architectural details; (iii) a compiler to a high-level language supports conveniently introducing tracers (Section 5.5), aiding in the debug cycle.

While our focus is on safety, we also took steps to ensure the correctness of our tools. Indeed, to aid in debugging vWasm during development, we implemented a semantics fuzzer, included in our open source release, which randomly generates valid Wasm programs that check their own results during execution, in order to help identify potentially flawed semantics. The core idea of this fuzzer is to generate code of the form "if 2 + 3 is not 5, exit with failure". This fuzzer helped us identify and fix over 15 distinct semantic correctness issues in vWasm (none of which threatened sandbox safety, but which could lead to incorrect computation results). No issues were found by the fuzzer in rWasm. Both vWasm and rWasm have now been fuzzed extensively, and they both pass all correctness and consistency checks for the benchmarks.

Overall, we conclude that the initial development effort for rWasm is significantly better than vWasm.

Development Effort: Maintenance Of course, the development effort for any software used in practice cannot be understood purely from its initial implementation effort but must also consider ongoing maintenance costs. This can be quite subtle for both vWasm and rWasm, and also somewhat speculative. For the former, there is no further maintenance effort needed if one simply wants to use it on x64 processors, since the architecture, while likely to change, will largely only introduce new instructions while keeping the existing instructions the same. However, improving the performance of code generated by vWasm would require quite some effort. Since the stages before the sandboxing pass are unverified, the effort is comparable to any other maintenance for a compiler. Wasm, as a standard, is not yet completely finished and new proposals will continue to improve upon it, adding new instructions and potentially new control-flow constructs; these should only require a small amount of effort to introduce. However, if Wasm introduces a new way to access memory, this could take a larger amount of effort to introduce into vWasm, since it might impact the sandboxing pass. Finally, despite our efforts to keep our proofs as general as possible, adding support for a new architecture (such as ARM) would be a straightforward but non-trivial amount of effort, given the inherent difficulty of writing formal proofs. In contrast, rWasm automatically supports all architectures supported by the Rust compiler; thus, support for new architectures comes to rWasm "for free" from the broader Rust community. Similarly, performance of the overall compiler automatically improves as `rustc`'s performance is improved, without any changes needed to rWasm's code itself. Additionally, new domain-specific optimizations, new Wasm instructions, control flow constructs, ways to access memory, etc. could be added to rWasm with little effort, as seen by the ease of the initial implementation.

Static Property Extensibility Provable safety is an important property of a verified sandboxing compiler, but one might wish to prove other properties, such as traditional compiler correctness. Here, vWasm has the upper hand, as this is feasible to do in F*, and we have even structured the compiler to make such proofs possible. In contrast, proving correctness for rWasm would be a challenging task, since one would need to formally model the Rust language, show that rWasm preserves Wasm semantics in compiling to Rust, and then implement a semantics-preserving Rust compiler (or prove `rustc` as semantics-preserving). The nature of the provable sandboxing property is what puts it into the sweet spot where we obtain it "for free" when compiling to Rust, and we believe there may be other such properties where one can obtain provable guarantees in a similar fashion. However, all these properties are a strict subset of what might be proven for an implementation like vWasm, which is built in a full-blown verification-oriented language.

Run-Time Extensibility As discussed in Section 5.5, rWasm supports conveniently adding runtime tracers, or Inline Reference Monitors (IRMs). It does so by leveraging its emulation of Wasm in a high-level language (Rust) to succinctly inspect and modify the program's state. In contrast, implementing these for vWasm would require a significant re-architecture of the compiler, which follows a traditional compiler pipeline oriented towards progressively lowering Wasm towards machine code. Thus, run-time behavior of a Wasm module can be better observed, analyzed, and controlled when compiled via rWasm.

7 Related Work

Virtualization-based Isolation Hypervisors and VMMs can provide strong sandboxing, such as with the Hypervisor-Protected Code Integrity (HVCI) [14] option for drivers on Windows. However, this is heavyweight and usually requires hardware support. Lighter weight than this, most operating systems guarantee strong isolation between processes, and some even provide OS-level virtualization, used by container frameworks such as Docker [6] and LXC [25]. This can still be too expensive due to the overhead of IPC, multiple privilege-level crossings, cache flushes, etc. Instead, SFI provides sandboxing that is intra-process, costing little more than a function call. To use a sandboxed module, a developer simply links against it, easing deployment.

Language-based Isolation Some programming languages have VMs that can provide isolation guarantees (e.g., V8 for JavaScript, JVM for Java, and CLR for .NET). However, these must trust the complex implementation of the language's VM, and they restrict usage to their particular language. In contrast, by using Wasm as a narrow waist (Section 2), we support sandboxing for nearly all popular languages.

Validator-based SFI Software Fault Isolation (SFI) [41] is a popular technique for providing language-agnostic, lightweight, and safe software sandboxing. Traditionally, SFI solutions have an untrusted component that introduces the checks, and a trusted validator that confirms that the checks are sufficient before execution. On the Web, NaCl [52] uses this approach, and RockSalt [29] even provides a formally verified validator. They rely, however, on a custom compiler toolchain to make the emitted code easier to validate. In contrast, VeriWasm [17] is a formally verified validator that confirms checks produced by an uncustomized compiler, Lucet [1], which further optimizes code *after* inserting SFI checks. To do this without false positives, VeriWasm uses features of Wasm, as well as implementation choices specific to Lucet. This is quite challenging to do without rejecting legitimate programs or suffering soundness issues (e.g., CVE-2021-32629), due to Rice's theorem [36]. Instead, our approaches guarantee sandbox safety by construction. Additionally, while validator-based approaches like NaCl, RockSalt and VeriWasm are necessarily architecture dependent, rWasm provides architecture-agnostic provable sandboxing. Finally, while most previous SFI solutions use a fixed-size sandbox for performance, rWasm attains high performance without reserving a large, fixed-sized chunk of memory (see Section 5.4), making it feasible to use even in embedded environments.

Compilation-based SFI Kroll et al. [21] present a technique for Portable SFI (PSFI) that is architecture-agnostic, with performance comparable to GCC [10] with no optimizations enabled. PSFI works on Cminor, an intermediate language of CompCert [24] (which compiles from C), and works by composing a program transformer with the verified backend of CompCert. Our compilers instead are multi-lingual since they support any language that can be compiled to Wasm (Section 2.3). In theory, PSFI could also be extended to offer multi-lingual support by writing translators from other languages to Cminor.

Additionally, rWasm obtains competitive performance with unverified performance-optimized implementations (Section 6.1). Due to Rust's collection of supported target architectures, rWasm can also target more architectures with no additional effort.

WebAssembly Multiple compelling use cases have been shown for using Wasm as a sandboxing primitive. RLBox [30] provides a framework for retrofitting isolation of third-party libraries in complex pre-existing software like Firefox; eWASM [32] provides a framework for SFI using Wasm on embedded systems with resource constraints; and Sledge [9] enables low-latency serverless compute on the edge via Wasm. Employing our techniques for provably-safe sandboxing within these framework would provide greater assurance of their safety.

Given Wasm's growing prevalence, it is important to identify its performance bottlenecks compared to running purely native code. Jangda et al. [16] perform a large-scale evaluation of browser Wasm runtimes, which helps identify causes for these bottlenecks, some inherent to Wasm, and others due to implementation deficiencies. These highlight opportunities that vWasm could take to improve performance.

Recent work by Lehmann et al. [23] shows that classic vulnerabilities such as simple stack buffer overflows, unexploitable in native binaries due to common mitigations, become exploitable again inside Wasm modules. This however does not impact sandboxing runtimes for Wasm such as ours, as any exploit will only corrupt the program state within the sandbox. The environment is left unaffected, modulo calls via the trusted interface offered explicitly by the environment.

Proposed extensions to Wasm, such as MS-Wasm [5] (for memory safety), can help capture critical high-level information about the program being compiled Wasm. Other extensions like CT-Wasm [35] (for constant-time cryptography) help capture high-level invariants that one wishes to maintain. These extensions are orthogonal to the goals of sandboxing, and can help provide even stronger guarantees of safety and correctness to the native code execution.

8 Conclusions

In this work, we have explored two concrete points in the design space for implementing a sandboxing execution environment, with a focus on WebAssembly. We proposed designs for these two points, implemented them as open-source tools, vWasm and rWasm, and evaluated them on a collection of both quantitative and qualitative metrics. We show that run-time performance and provable safety are not in conflict, and indeed rWasm is the first Wasm runtime that is both provably-sandboxed and fast.

Acknowledgments

We would like to thank Deian Stefan for helpful comments and suggestions on an early draft of this paper. We would also like to thank our anonymous reviewers for their feedback.

This work was supported by a fellowship from the Alfred P. Sloan Foundation and by the CONIX Research Center, one of six centers in JUMP, a Semiconductor Research Corporation (SRC) program sponsored by DARPA.

References

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

[2] Jay Bosamiya, Sydney Gibson, Yao Li, Bryan Parno, and Chris Hawblitzel. Verified transformations and Hoare logic: Beautiful proofs for ugly assembly language. In *In Proceedings of the Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)*, July 2020.

[3] Clang: a C language family frontend for LLVM. https://clang.llvm.org/, 2020.

[4] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[5] Craig Disselkoen, John Renner, Conrad Watt, Tal Garfinkel, Amit Levy, and Deian Stefan. Position paper: Bringing memory safety to WebAssembly. In *Hardware and Architectural Support for Security and Privacy (HASP)*. ACM, June 2019.

[6] Docker. https://www.docker.com/, 2021.

[7] Úlfar Erlingsson and Fred B. Schneider. SASI enforcement of security policies: A retrospective. In *Proceedings of the 1999 Workshop on New Security Paradigms*, NSPW '99, page 87–95, New York, NY, USA, 1999. Association for Computing Machinery.

[8] Ethereum WebAssembly (ewasm). https://ewasm.readthedocs.io/en/mkdocs/, 2021.

[9] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. Sledge: A serverless-first, light-weight Wasm runtime for the Edge. In *Proceedings of the 21st International Middleware Conference*, Middleware '20, page 265–279, New York, NY, USA, 2020. Association for Computing Machinery.

[10] GCC, the GNU Compiler Collection. https://gcc.gnu.org/, 2020.

[11] 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. *SIGPLAN Not.*, 52(6):185–200, June 2017.

[12] Norm Hardy. The confused deputy: (or why capabilities might have been invented). *SIGOPS Oper. Syst. Rev.*, 22(4):36–38, October 1988.

[13] How many x86 instructions are there? https://fgiesen.wordpress.com/2016/08/25/how-many-x86-instructions-are-there/, 2016.

[14] Hypervisor-Protected Code Integrity (HVCI). https://docs.microsoft.com/en-us/windows-hardware/drivers/bringup/device-guard-and-credential-guard.

[15] Intel®. Intel® 64 and IA-32 Architectures Software Developer's Manual Combined Volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4, April 2021.

[16] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. Not so fast: Analyzing the performance of WebAssembly vs. native code. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 107–120, Renton, WA, July 2019. USENIX Association.

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

[18] Ralf Jung, Jacques-Henri Jourdan, Robbert Krebbers, and Derek Dreyer. RustBelt: Securing the foundations of the Rust programming language. *Proc. ACM Program. Lang.*, 2(POPL), December 2017.

[19] Zijo Kenjar, Tommaso Frassetto, David Gens, Michael Franz, and Ahmad-Reza Sadeghi. V0LTpwn: Attacking x86 processor integrity from software. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1445–1461. USENIX Association, August 2020.

[20] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture*, ISCA '14, page 361–372. IEEE Press, 2014.

[21] Joshua A. Kroll, Gordon Stewart, and Andrew W. Appel. Portable software fault isolation. In *2014 IEEE*

27th Computer Security Foundations Symposium, pages 18–32, 2014.

[22] Ramana Kumar, Magnus O. Myreen, Michael Norrish, and Scott Owens. CakeML: a verified implementation of ML. In *Proceedings of the ACM Symposium on Principles of Programming Languages (POPL)*, January 2014.

[23] Daniel Lehmann, Johannes Kinder, and Michael Pradel. Everything old is new again: Binary security of WebAssembly. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 217–234. USENIX Association, August 2020.

[24] Xavier Leroy, Sandrine Blazy, Daniel Kästner, Bernhard Schommer, Markus Pister, and Christian Ferdinand. CompCert – a formally verified optimizing compiler. In *Embedded Real Time Software and Systems (ERTS)*. SEE, 2016.

[25] Linux Containers. https://linuxcontainers.org/, 2021.

[26] Nicholas D. Matsakis and Felix S. Klock. The Rust language. In *Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology*, HILT '14, page 103–104, New York, NY, USA, 2014. Association for Computing Machinery.

[27] Stephen McCamant and Greg Morrisett. Evaluating SFI for a CISC architecture. In *15th USENIX Security Symposium (USENIX Security 06)*, Vancouver, B.C. Canada, July 2006. USENIX Association.

[28] Tyler McMullen. Lucet: A Compiler and Runtime for High-Concurrency Low-Latency Sandboxing. Talk presented at the Principles of Secure Compilation (PriSC) workshop. https://youtu.be/WddPA0U6v2A?t=2309s, 2020.

[29] Greg Morrisett, Gang Tan, Joseph Tassarotti, Jean-Baptiste Tristan, and Edward Gan. RockSalt: Better, faster, stronger SFI for the x86. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 395–404, New York, NY, USA, 2012. Association for Computing Machinery.

[30] 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 *29th USENIX Security Symposium (USENIX Security 20)*, pages 699–716. USENIX Association, August 2020.

[31] Authors of VeriWasm. Private Correspondence.

[32] Gregor Peach, Runyu Pan, Zhuoyi Wu, Gabriel Parmer, Christopher Haster, and Ludmila Cherkasova. eWASM: Practical software fault isolation for reliable embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 39(11):3492–3505, 2020.

[33] PolyBench-C: the Polyhedral Benchmark suite. https://web.cs.ucla.edu/~pouchet/software/polybench/. Accessed: January 2021.

[34] Aseem Rastogi, Guido Martínez, Aymeric Fromherz, Tahina Ramananandro, and Nikhil Swamy. Layered indexed effects: Foundations and applications of effectful dependently typed programming. https://www.fstar-lang.org/papers/layeredeffects/, 2020.

[35] John Renner, Sunjay Cauligi, and Deian Stefan. Constant-time WebAssembly. In *Principles of Secure Compilation (PriSC)*, January 2018.

[36] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical Society*, 74(2):358–366, 1953.

[37] 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 *19th USENIX Security Symposium (USENIX Security 10)*, Washington, DC, August 2010. USENIX Association.

[38] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. AddressSanitizer: A fast address sanity checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, Boston, MA, June 2012. USENIX Association.

[39] Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *Proceedings of the ACM Conference on Principles of Programming Languages (POPL)*, 2016.

[40] The Rust programming language. https://www.rust-lang.org/, 2021.

[41] Robert Wahbe, Steven Lucco, Thomas E. Anderson, and Susan L. Graham. Efficient software-based fault isolation. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, SOSP '93, pages 203–216, New York, NY, USA, 1993. ACM.

[42] WASI – The WebAssembly System Interface. https://github.com/WebAssembly/WASI, 2021.

[43] wasm2c. https://github.com/WebAssembly/wabt, 2021.

[44] wasm3. https://github.com/wasm3/wasm3.

[45] Wasmer - The Universal WebAssembly Runtime. https://wasmer.io/, 2021.

[46] Wasmtime: A small and efficient runtime for WebAssembly & WASI. https://wasmtime.dev/, 2021.

[47] Conrad Watt. Mechanising and verifying the WebAssembly specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs*, CPP 2018, page 53–65, New

York, NY, USA, 2018. Association for Computing Machinery.

[48] WAVM: WebAssembly virtual machine. https://wavm.github.io/, 2021.

[49] WebAssembly Micro Runtime (WAMR). https://github.com/bytecodealliance/wasm-micro-runtime, 2021.

[50] Aaron Weiss, Daniel Patterson, Nicholas D. Matsakis, and Amal Ahmed. Oxide: The essence of Rust. *CoRR*, abs/1903.00982, 2019.

[51] Hui Xu, Zhuangbin Chen, Mingshen Sun, Yangfan Zhou, and Michael Lyu. Memory-safety challenge considered solved? An in-depth study with all Rust CVEs, 2021.

[52] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. Native client: A sandbox for portable, untrusted x86 native code. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 79–93. IEEE, 2009.

Appendix A Normalized Execution Time Per Benchmark

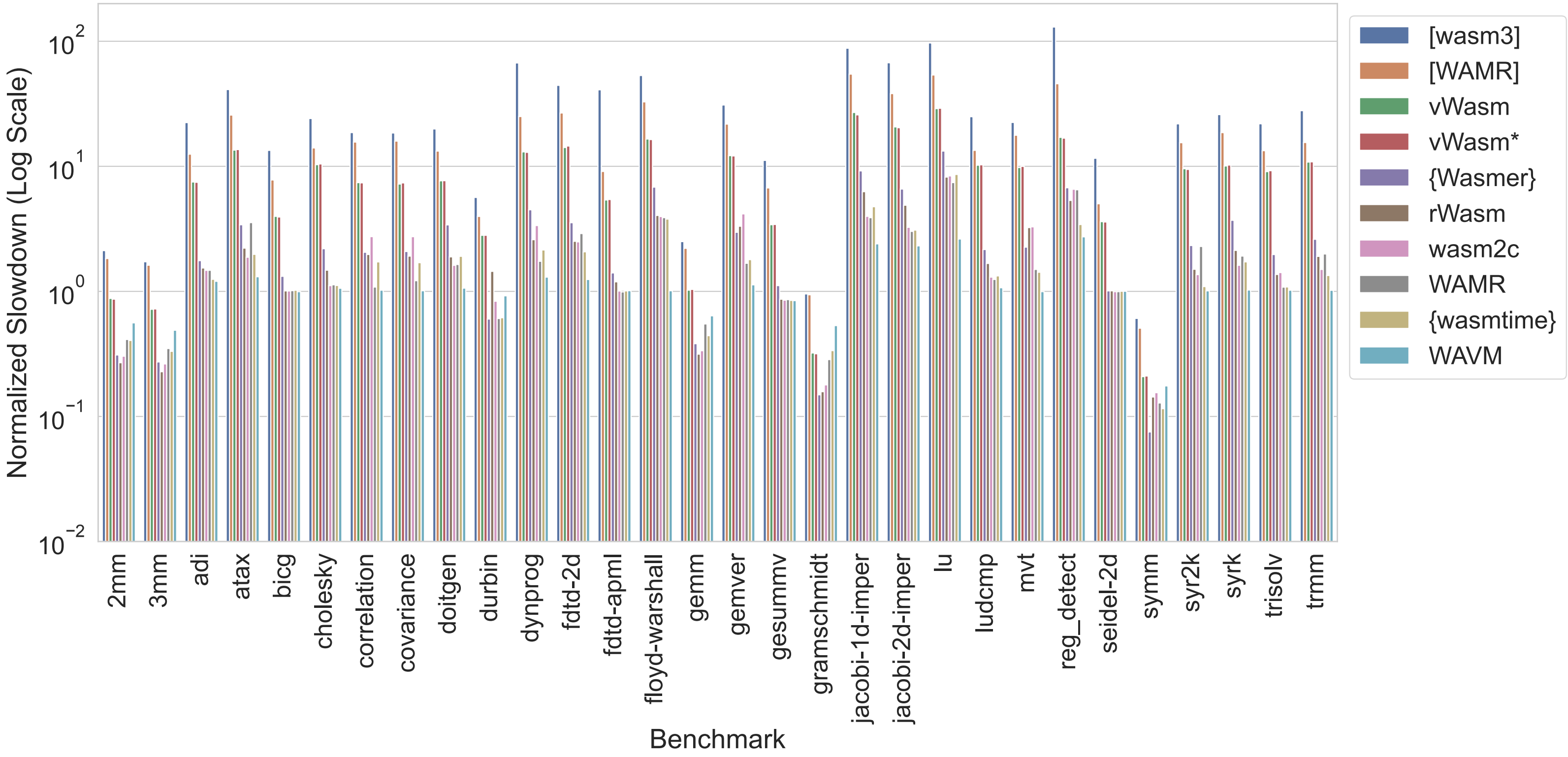

Figure 9: Per-benchmark breakdown of the mean execution time of PolyBench-C benchmarks across the WebAssembly runtimes, normalized to pure native execution. Interpreters have square brackets; JIT compilers have braces; the rest are AOT compilers. vWasm* disables sandboxing. All benchmarks are compiled with PolyBench-C's own internal execution time reporting (i.e., `-DPOLYBENCH_TIME`), rather than relying on less accurate external measurements using `time(1)` or similar. Note how, for example, all interpreters and compilers on `lu`/`symm` perform strictly worse/better than native. Such differences have been seen in the past [11], and seem connected to how well compilation to Wasm goes, as well as the memory access patterns of each benchmark.