WaVe: a verifiably secure WebAssembly sandboxing runtime

Evan Johnson† Evan Laufer‡ Zijie Zhao⋆ Dan Gohman⋄
Shravan Narayan† Stefan Savage† Deian Stefan† Fraser Brown∗
†UC San Diego ‡Stanford ⋆UIUC ⋄Fastly Labs ∗CMU

Abstract—**The promise of software sandboxing is flexible, fast and portable isolation; capturing the benefits of hardware-based memory protection without requiring operating system involvement. This promise is reified in WebAssembly (Wasm), a popular portable bytecode whose compilers automatically insert runtime checks to ensure that data and control flow are constrained to a single memory segment. Indeed, modern compiled Wasm implementations have advanced to the point where these checks can themselves be verified, removing the compiler from the trusted computing base. However, the resulting integrity properties are only valid for code executing *strictly* inside the Wasm sandbox. Any interactions with the runtime system, which manages sandboxes and exposes the WebAssembly System Interface (WASI) used to access operating system resources, operate outside this contract. The resulting conundrum is how to maintain Wasm's strong isolation properties while still allowing such programs to interact with the outside world (i.e., with the file system, the network, etc.). Our paper presents a solution to this problem, via WaVe, a verified secure runtime system that implements WASI. We mechanically verify that interactions with WaVe (including OS side effects) not only maintain Wasm's memory safety guarantees, but also maintain access isolation for the host OS's storage and network resources. Finally, in spite of completely removing the runtime from the trusted computing base, we show that WaVe offers performance competitive with existing industrial (yet unsafe) Wasm runtimes.**

1. Introduction

WebAssembly (Wasm) is a portable bytecode designed to run everywhere at near-native speeds [1], [2]. Unlike most other bytecodes, Wasm was designed with safety in mind from the start: Wasm code runs in a sandboxed environment, because the compiler (or interpreter) inserts runtime checks that restrict the code to its own region of memory. As a result, Wasm has become popular beyond its original home, the Web; it has been used to isolate code on edge clouds [3], [4], embedded devices [5], blockchains [6], browsers [7], [8], and network proxies [9].

At the surface, Wasm's safety guarantees are clear: a Wasm program is isolated to its own sandboxed region of memory as long as the Wasm compiler (or interpreter) inserts safety checks in all the right places. Getting safety checks *right*, then, has become a focus of serious effort. Researchers have fuzzed Wasm compilers [10], [11], verified the safety of Wasm binaries [12], and built verified Wasm compilers [13] and interpreters [14]. Unfortunately, enforcing strong memory isolation alone is not enough.

In practice, every Wasm program relies on a runtime system—and today, that runtime system is trusted. It's trusted to correctly set up and tear down Wasm sandboxes, which involves low-level memory management code that's error-prone even in memory-safe languages like Rust [15], [16]. More significantly, the runtime is trusted to correctly implement the WebAssembly System Interface (WASI) [17], the interface Wasm programs use to read files, make network requests, and otherwise access operating system resources. Implementing WASI is also extremely tricky.

For one, WASI was based on (a subset of) POSIX [18] and so inherits many of POSIX's flaws; as one example, WASI, like POSIX, is written in informal English. This means that WASI runtimes must implement an interface *from* informal semantics, and *for* different OS platforms—each with slightly different, similarly informal specifications [19].

Another challenge in implementing WASI is correctly enforcing security. First, the runtime must enforce Wasm's *memory isolation* end-to-end, i.e., ensure that every *hostcall* into the runtime (and, in turn, every syscall into the underlying OS) on behalf of one sandbox does not read or clobber memory that belongs to another sandbox or the runtime itself. Second, the runtime must enforce WASI's *resource isolation* policies, which restrict the sandbox's access to resources like the file system and network. To do this, developers must essentially replicate parts of the kernel's logic (e.g., how paths are resolved)—which they do by wrapping system calls with low-level glue code and security checks. In practice, they often get it wrong [20], [21], [22], [23], [24], [25] and, unfortunately, this is not surprising: there's a history of bugs in similar sandbox runtime systems, and a history of attackers exploiting those bugs [26], [27], [28].

In response to this problem, we built WaVe, a new Wasm runtime system that is verifiably secure, fast, and WASI-compliant across POSIX platforms. At its core, WaVe uses automated verification to ensure that the runtime code, even when calling into the underlying OS, preserves Wasm's memory isolation guarantees and correctly restricts each sandbox's access to OS resources like the filesystem and network. Two insights make WaVe work in practice.

First, we believe that security policies (e.g., memory isolation, filesystem isolation, and network isolation) should be explicit and decoupled from enforcement. This not only makes it clear (and easy to audit) which policies WaVe enforces but also ensures that WaVe enforces a uniform policy across all WASI hostcalls for all target operating systems. This also makes it easy to safely extend the runtime with new functionality. For example, beyond the core set of WASI hostcalls, WaVe exposes the networking hostcalls described in the WASI-sockets proposal [29] and enforces the proposal's described safety policy.

Our second insight is to only model OS semantics in as much detail as necessary to capture any system call's effects on security. For example, WaVe's model of the POSIX `read(fd,buf,count)` syscall does not model kernel data structures (e.g., file descriptor tables, inodes, or buffer caches); instead it only models its impact on memory isolation—`read` may write `count` bytes starting at `buf`—and its impact on filesystem isolation—`read` may read from file (descriptor) `fd`. This makes it possible to "pay as you go" (e.g., you don't need to specify details about file descriptors if you only care about memory isolation). It also makes it possible for us to reuse specifications across operating systems. For example, though POSIX semantics actually vary across operating systems [19], our abstract semantics make it possible for WaVe to largely share specifications across Linux and MacOS.

We implement WaVe (with support for WASI on Linux and MacOS) in roughly 7.3K lines of Rust. Most of this code is untrusted and verified using Prusti [30]; the code that *is* trusted (beyond the verification tool itself) is specification code—the Wasm/WASI security policy (43 lines of code) and OS specifications (567 lines). While this code is short enough to be auditable, we use host- and system-call fuzzing to evaluate the correctness of our specification (and thus the security of WaVe); our fuzzing effort didn't reveal any bugs.

We also evaluate WaVe on dimensions other than security: functional correctness, portability, and performance. First, we differentially fuzz WaVe against four popular runtimes, including the Wasmtime [31] used in production at Fastly; we find that our implementation of WASI is consistent with these runtimes on both Linux and MacOS. And, second, we measure the performance overhead of WaVe relative to Wasmtime on micro-benchmarks (e.g., LMbench [32]) and macro-benchmarks (e.g., SQLite [33] and SPEC CPU [34]). We find that WaVe's performance is comparable to—and often better than—Wasmtime's, while also providing stronger (verifiable) security guarantees.

Open source All source code, including WaVe and our benchmarks, are available at `wave.programming.systems`.

2. Overview

In this section, we give background on WASI runtimes, and describe how bugs in system interfaces can break language-level sandboxing guarantees. Then, we introduce WaVe and describe how it prevents isolation-breaking bugs by construction.

2.1. The WASI runtime

WASI's goal is to expose a system interface to Wasm code that is flexible enough to implement standard libraries like libc and portable enough to run across different operating systems. WASI accomplishes this by providing low level OS-agnostic APIs (known as hostcalls) to talk to the underlying operating system—defining, for example, hostcalls for accessing the filesystem and network [35]. These hostcalls are exposed to potentially malicious code running inside the Wasm sandbox, and WASI runtimes enforce a common safety policy described in the WASI spec to limit the resources exposed to sandboxed code.

WASI's safety guarantees WASI implementations are expected to make three guarantees about how the sandbox interacts with the OS: they must guarantee memory isolation, file system isolation, and network isolation (Figure 1).

Memory isolation ensures that all memory reads and writes performed on behalf of the sandbox operate on memory that belongs to the sandbox (i.e., lies within the sandbox's linear memory). *File system isolation* ensures that each file the sandbox accesses is within the sandbox's assigned root directory. For example, a sandbox assigned `/foo/bar` as its root directory would be able to access `/foo/bar/data.txt` and `/foo/bar/img.png` but not `/foo/secret.txt`. *Network isolation* ensures that each socket created by the sandbox belongs to known and explicitly allowed network protocols (e.g., the sandbox only uses TCP and UDP). It also ensures that these sockets only connect to addresses specified by the host application in an allow-list that cannot be modified by the sandbox.

It is the responsibility of the WASI runtime to enforce these safety guarantees for all hostcalls exposed to the sandbox.

Hostcalls enforce WASI's safety guarantees Hostcalls are responsible for checking the safety of hostcall parameters and translating them from the sandbox to the OS. They must perform this check-and-translate process such that any interactions with the underlying operating system do not violate the runtime's isolation invariants. Consider, for example, the `path_remove_directory` hostcall in WASI:

```
fn path_remove_directory(sbox: &VmCtx,
    dirfd: i32, path: SboxPtr, path_len: u32) {...}
```

This WASI hostcall deletes the directory at `path`, relative to file descriptor `dirfd` (i.e., if `dirfd` referred to a directory at `/foo/bar` and `path` is `./baz`, this call should remove `/foo/bar/baz`). To implement `path_remove_directory` correctly, the runtime must:

- Check that the buffer specified by `path` and `path_len` is entirely within the sandbox's allocated memory.
- Check that `path` is within the sandbox's root directory—making sure to account for symbolic links.
- Translate `path` from the sandbox's string representation to the OS's representation and convert it to a path relative to the sandbox's root directory.
- Invoke the OS with the correct set of flags (on POSIX, `path_remove_directory(sbox, dirfd, path, path_len)` calls `unlinkat(host_dirfd, host_path, AT_REMOVEDIR)`).
- Translate the system call return error code from the OS's representation to the WASI representation.

And, at each step, the runtime needs to account for portability across all POSIX systems.

Unsurprisingly, each steps is also error prone: memory bounds checks may contain integer overflows and off-by-one errors, path translation may mishandle OS semantics (e.g.,

Property	Guarantee
Memory isolation	All memory accesses performed on behalf of the sandbox lie within the its linear memory.
Filesystem isolation	Each file the sandbox accesses is within the its assigned root directory.
Network isolation	Each socket the sandbox creates belongs to a known and explicitly allowed protocol. These sockets only connect to addresses specified in an allow-list that the sandbox cannot modify.

Figure 1: Safety properties WaVe enforces and their security guarantees.

```
1 bool SharedMemory::Invoke(...) {
2   ...
3   if (NULL == shared_memory->map_addr_ ||
4       offset + len > shared_memory->size_) {
5     NULL_TO_NPVARIANT(*result);
6     return false;
7   } else // copy len bytes into shared memory
8 }
```

Figure 2: Memory safety bug in the NaCl runtime.

```
1  fn path_remove_directory(sbox: &WasiSbox,
2    dirfd: __wasi_fd_t, path: WasmPtr, path_len: u32) {
3    // Copy path from the sandbox to the runtime
4    let path_str = sbox.get_sbox_str(path, path_len);
5  ...
6    // Check and translate the path
7    let host_path = validate_host_path(dirfd, path_str);
8  ...
9    // Invoke the OS to remove the directory at path
10   state.fs_remove_dir(path_str);
11 }
```

Figure 3: Filesystem isolation bug in the Wasmer runtime (simplified).

not taking symbolic links into account), and system call invocation may misuse the POSIX interface by using the wrong combination of flags to uphold the runtime's safety policy. And since existing runtimes rely on ad-hoc policy enforcement, the developer is responsible for placing all the right checks in all the right places. Understandably, they do not always do so.

2.2. Unsafety in sandboxing runtimes

Unsafety in the runtime can take one of two forms: the runtime can inadvertently violate the sandbox's memory isolation guarantees, or the runtime can incorrectly limit the sandbox's access to OS resources like the file system or network. Memory safety bugs and resource isolation bugs can both lead to sandbox escapes.

Violating memory isolation in the runtime To understand the implications of memory isolation bugs in system interfaces, consider the C++ snippet from the NaCl toolkit in Figure 2. This code implements an interface for accessing shared memory across the sandbox boundary. The function takes an `offset` parameter—the offset within the shared memory—and a `len` parameter—the number of bytes to read (or write). On lines 3 and 4, the function checks that the access is within the memory region and, if so, it copies `len` bytes into the shared memory (on line 7). Though these checks look reasonable at first glance, they are incorrect: a sufficiently large sum of `offset` and `len` may overflow to a value that is smaller than `shared_memory->size`. This allows an attacker to access memory beyond the shared memory bounds, including memory that belongs to the runtime or to other sandboxes [36].

We chose an example from NaCl to stress that this problem is also present in non-Wasm SFI systems. Similar errors occur in industrial Wasm runtimes as well [37]. Indeed, we found similar integer overflow bugs in both the WAMR and Wasmer runtimes [38], [25], [39], [40].

While memory safety errors are primarily a problem in runtimes written in C/C++ (as many runtimes are [5], [41], [42]), they can also occur in memory safe languages when the runtime interacts with unsafe interfaces like the system call interface. This is because system calls are not aware of application-level safety policies (like subprocess isolation schemes) and will happily read or write to any memory location, regardless of whether the application considers it safe or not. System calls are also not aware of what OS resources the sandbox is permitted to access, and bugs in the runtime's resource isolation enforcement can lead to sandbox escapes.

Violating resource isolation in the runtime WASI runtimes are not just expected to prevent memory isolation bugs, they are also required to prevent resource isolation bugs. For example, WASI runtimes must enforce that sandboxes only access files within the sandbox's assigned root directory. And even when runtimes implement safety checks correctly, they may suffer from inconsistent policy enforcement since they rely on ad-hoc application of these checks.

Consider the `path_remove_directory` hostcall [43] as implemented in the Wasmer [44] runtime (shown in Figure 3). On line 4, the hostcall reads a path from the sandbox. On line 7, the hostcall validates the path by checking that it lies within the sandbox's root directory, and rewrites the path to its location on the host operating system, relative to the sandbox's root directory. However, on line 10, the hostcall invokes the OS to delete the unchecked path provided by the sandbox. This lets the sandbox delete files outside its root directory, breaking the runtime's promise of filesystem isolation.

This path validation and translation procedure is correctly applied in eight other hostcalls. But here it's not. This oversight is a symptom of the ad-hoc policy enforcement that current sandboxing runtimes rely on—it is the developer's responsibility to put all the right checks in all the right places. Instead of placing the burden of correctness on the developer, we can use an automatic verifier to check that the runtime's isolation policy holds true—and remove this runtime code

```
1  fn wasi_path_remove_directory(ctx: &mut VmCtx,
2    dirfd: u32, path: u32, path_len: u32) -> Result<()> {
3     if !(path+path_len < ctx.memlen && path <=
   ↪    path+path_len) {
4       return Err(EFault);
5     }
6  ...
7     let mut pathname: Vec<u8> = Vec::new();
8     pathname.reserve_exact(path_len as usize);
9     // VERIFIER CHECKS:
10    // (1): pathname.len() == path_len as usize
11    // (2): (ctx.mem+path) + (ctx.mem+path_len) <
   ↪    (ctx.mem+ctx.memlen) && (path <= path+path_len)
12    memcpy(pathname, ctx.mem + path, path_len);
13 ...
14    let os_dirfd = ctx.translate_fd(dirfd)?;
15    let host_path = ctx.translate_path(pathname, false,
   ↪    fd)?;
16    // VERIFIER KNOWS:
17    // ctx.is_in_sbox_fs(host_path)
18 ...
19    // VERIFIER CHECKS:
20    // ctx.is_in_sbox_fs(host_path)
21    let res = unlinkat(ctx, os_dirfd, host_path,
   ↪    AT_REMOVEDIR);
22    return translate_errno(res);
23 }
```

Figure 4: WaVe's implementation of the `path_remove_directory` hostcall. For clarity, the code is annotated with the conditions that WaVe checks to prevent bugs—these annotations do not need to be written by the developer.

from the TCB entirely. WaVe runtime does precisely this.

2.3. Securing the runtime with WaVe

WaVe is a verifiably secure WASI runtime. WaVe addresses the problems current WASM runtimes face—enforcing policies ad hoc, mishandling complex OS semantics, and misusing the POSIX interface—by maintaining WASI's safety guarantees with the help of an automatic verifier. Instead of relying on implicit developer knowledge about POSIX and OS semantics, WaVe uses an explicit OS specification that describes the effects that each syscall has on userspace memory and the OS. Instead of relying on developers to put all the right safety checks in all the right places, WaVe uses a single centralized, auditable, and testable (§6) safety policy that it enforces with an automatic verifier. In total, WaVe enforces memory isolation, file system isolation, and network isolation (described in the safety policy) by statically proving that system calls are used safely.

Consider the two bugs from Section 2.2—the memory isolation bug in NaCl caused by an integer overflow in a bounds check, and the filesystem isolation bug in Wasmer caused by missing checks. WaVe can catch both these types of bugs by statically checking preconditions on potentially unsafe runtime operations (i.e, using `memcpy` to copy code to/from the sandbox, and allowing the sandbox to access the file system).

To understand how WaVe prevents these classes of bugs, consider WaVe's implementation of the WASI `path_remove_directory` hostcall, shown in Figure 4. The code is annotated with the conditions that WaVe checks to prevent such bugs; note that *these annotations are added purely for clarity*, and the conditions WaVe checks are actually described just once in the safety policy file.

On lines 3–5, the hostcall performs a bounds check on the `path` and `path_len` parameters to ensure that the path buffer is within the sandbox's memory. It checks that `path+path_len` is less than the upper bound of sandbox memory (a lower bound is not necessary since `path` is an unsigned offset into the sandbox's linear memory), and that `path+path_len` does not overflow. Later, on line 12, the hostcall goes to `memcpy` the data out of the sandbox. At the `memcpy`, the WaVe verifier statically checks that the buffer is within sandbox memory, does not overflow, and fits into the destination buffer; see the `VERIFIER CHECKS` comment on line 4. Had the runtime code on lines 3–5 not checked for integer overflow, the verifier would register an error.

On line 15, the sandbox translates `pathname` from the sandbox's string representation to the OS's representation, and checks that `pathname` is within the sandbox's root directory. After this call, the verifier knows that `host_path` is within the sandbox's root directory (line 22); that's because `ctx.is_in_sbox_fs(host_path)` is the developer-written postcondition of the `translate_path` method. All the developer must do is *write* the postcondition; the verifier automatically checks that the implementation of `translate_path` enforces that postcondition.

On line 21 the hostcall invokes the OS to remove the directory. Here, the verifier recognizes that `unlinkat` is reading and writing to OS state (per the OS specification), so it statically checks that `host_path` is within the sandbox's root directory. It is! But if the code were buggy—and the verifier could not guarantee that `host_path` is within the sandbox's root directory, the verifier would throw an error and prevent the buggy code from even compiling.

3. Design

In this section, we describe the design of WaVe, a verifiably secure WASI-compliant Wasm runtime. First, we describe our threat model for malicious sandbox code attempting to compromise the runtime (including other sandboxes). Then, we describe WaVe's design and implementation, and give an overview of how it uses an automatic verifier to prove safety against sandbox escapes. Finally, we explain in more detail how WaVe models unsafe behavior in the sandboxing runtime using *effects*: the potentially unsafe actions performed on behalf of the sandbox that can inspect or modify runtime and OS state.

3.1. Threat model

Our assumptions reflect the threat model of industrial Wasm runtimes. We assume that sandboxed code is malicious and will try to escape the sandbox by making a sequence of arbitrary hostcalls with arbitrary arguments. We don't consider sandbox escapes caused by data corruption or

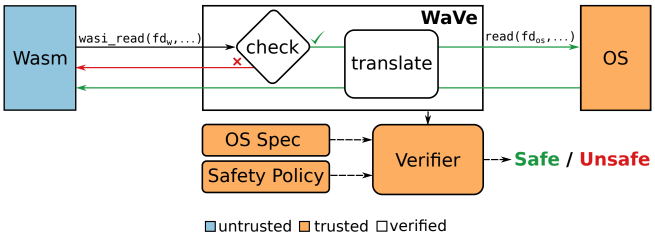

Figure 5: WaVe mediates the sandbox's requests for OS resources (hostcalls) via WASI. Given an OS specification and a safety policy, WaVe statically verifies that it only executes safe hostcalls.

control-flow-hijacking within the sandbox; several tools [12], [14], [13] already eliminate this attack vector.

We assume that each sandbox is allocated—and has exclusive access to—a *root directory* that contains all the data the sandbox is permitted to access, and an allow-list describing the network addresses the sandbox is permitted to access. We also assume that the sandbox's root directory only contains regular files, symbolic links, and trusted device files (i.e., no files with special semantics like `/proc/self/mem`). Finally, we assume only a single thread per sandbox, and assume that all functions exposed by WASI are synchronous; this is in line with the Wasm and WASI standards, respectively. Host applications may be running multiple sandboxes concurrently, though, so each sandbox has its own unique runtime state (in WaVe as in industrial runtimes).

3.2. WaVE's design

WaVe is a Wasm runtime that implements the WebAssembly System Interface: it exposes the 45 WASI-specified hostcalls to the sandbox, which the sandbox invokes when it requires access to OS resources like the file system or network. WaVe also manages the sandbox-specific state used to execute the hostcalls, such as the list of open file descriptors for a sandbox.

When a sandbox makes a hostcall, WaVe (like other Wasm runtimes) dynamically checks that the hostcall's arguments are safe (e.g., all pointers are within sandbox memory); then, WaVe translates these arguments to the host OS's representation, invokes the OS, and finally translates the return value back to the sandbox representation. This process (Figure 5) is standard for WASI implementations. What *isn't* standard is that **WaVe statically verifies that this check-and-translate process correctly enforces the runtime's safety policy.**

Conceptually, a hostcall is safe if it preserves the properties that the runtime is attempting to verify; in WaVe, those properties are currently memory safety, filesystem isolation, and network isolation. Mechanically, a hostcall is safe if it sufficiently constrains all potentially unsafe operations defined in an OS specification such that the guarantees in the safety policy hold. We describe our design goals, OS specification, and safety policy in more detail next.

Design goals Wasm runtime designers face two challenges when trying to safely implement WASI: (1) the syscalls that the runtime interacts with are informally specified, and specs and syscalls vary across different architectures; (2) the code to enforce Wasm's safety guarantees is sprinkled throughout the runtime—and forgetting to, for example, check if pointers are within the sandbox at a single `memcpy` can break safety. We thus designed WaVe with three goals in mind:

1) **Provide two centralized, explicit specifications**, one of the safety policy and one of syscall behavior. These specifications make it clear both what the runtime must enforce and what each individual syscall does.
2) **Pay only for what we prove**, since exhaustively specifying syscall behavior is difficult and time-consuming. In contrast, in WaVe, developers only specify the syscall behavior that's relevant to their safety policy.
3) **Decouple** the OS specification from the safety policy, and the policy specification from its enforcement. Instead of specifying correctness at each syscall—like the runtime *enforces* correctness at each syscall—the WaVe safety policy exists in only one place. The OS spec and the safety policy are also independent, which makes it easy to test the spec (§6) and re-use both the spec and the policy across different runtimes.

We describe OS specification and safety policy next.

The OS specification The OS specification file contains a specification for each system call WaVe uses to implement the 45 WASI hostcalls, e.g., for `close(fd)`, it specifies that the runtime accesses a file descriptor on the host file system (§3.3). Specifications are over *effects*, potentially unsafe actions (e.g., memory writes) that the runtime tracks in an *effects trace* (see Section 3.3). WaVe supports both Linux and MacOS, so when the syscall interface differs between the two OSes, WaVe uses two separate specs—one for Linux and one for MacOS. In this way, the OS specification is easy to extend to new operating systems: the developer only needs to add specifications for the system calls that differ from those of existing, supported OSes.

The safety policy WaVe's safety policy is centralized into a single auditable policy file. The policy is expressed in terms of a set of constraints on the trace of effects (§3.3) written in plain Rust. At verification-time, WaVe attempts to prove that every effect that hostcalls could cause adheres to this safety policy; if not, WaVe throws an error.

WaVe's safety policy follows the "pay as you prove" model: proving properties like file system isolation and network isolation are independent, so developers only need to prove the properties they use (although proving memory safety is prerequisite to proving any other property). The policy spec is also easy to extend: all the developer needs to do is add an extra constraint in this policy file; the automatic verifier will ensure that this policy holds everywhere in the runtime, or reject it if the policy doesn't hold.

Runtime function contracts Though WaVe attempts to automatically verify the safety policy, it sometimes asks the developer to annotate runtime functions with preconditions and postconditions (written in plain Rust). The developer is only responsible for declaring a pre/postcondition contract, not proving it: the verifier analyzes the runtime code to ensure that each contract is valid. If an implementation does not uphold its contract—or if a contract's postcondition isn't

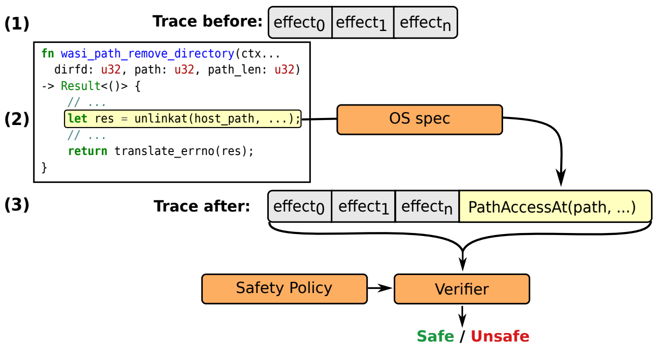

Figure 6: At verification time, WaVe checks that if a safety policy holds before each hostcall, the safety policy holds after. WaVe does this by proving that: given an safe effects trace at hostcall entry **(1)**, when the effects the hostcall causes (according to the OS spec) are appended to the trace **(2)**, that the trace is still safe at hostcall exit **(3)**.

strong enough to prove the safety policy—the verifier alerts the developer to try again. In practice, the annotation burden is small: WaVe has ≈ 300 lines of these contracts for its 4646 lines of verified code, and they are largely repetitive (e.g., many of them simply record that a buffer being passed around is within sandbox memory).

3.3. Modeling unsafe behavior with effects

In this section, we describe how WaVe models unsafety in the runtime in order to statically verify that all hostcalls respect safety policies; Section 4 details the proof of safety for memory safety, filesystem isolation, and network isolation.

WaVe models potentially unsafe actions as (extensible) *effects*; for example, a memory write could unsafely overwrite safety-critical data in the runtime (e.g., a function pointer), so WaVe tracks writes as effects. WaVe records a list of every effect that could occur during the execution of each hostcall in the *effects trace*. Specifications for syscalls and unsafe runtime functions like `memcpy` describe how these functions alter the effects trace. For each hostcall, WaVe guarantees that *if the trace was safe upon entering the hostcall, it is safe after the hostcall*. Figure 6 shows how WaVe proves this guarantee: when the verifier inspects a statement that is defined in the OS spec, it appends the effect described by the spec onto the trace. For example, when the verifier inspects the `path_remove_directory` function and reaches the `unlinkat` system call, it appends a `PathAccessAt` effect that describes every possible `path` that the system call could access. At the function exit, the verifier proves that for all possible values of `path`, the trace is still safe, and if it isn't, the verifier throws an error.

Next, we describe WaVe's effects, effects trace, and effect specifications in more detail; we also describe how to model new effects in order to verify more properties. Finally, we sketch how WaVe uses the trace to verify safety properties.

Effects WaVe proves safety, with respect to a given policy, by verifying that the runtime sufficiently constrains the *effects* that the sandbox can have. WaVe models the effects necessary to verify its three safety policies. For example, WaVe enforces

```
enum Effect {
    MemRead(addr, len),
    MemWrite(addr, len),
    FdAccess(fd),
    PathAccessAt(dirfd, path, follows_symlinks),
    SockCreation(domain, socktype, protocol),
    Shutdown(fd),
    NetConnect(ip, port),
}
```

Figure 7: The effects WaVe uses to model potentially dangerous behaviors.

a property called *filesystem isolation* (§4.2) wherein the runtime constrains a sandbox to a single directory which acts as the sandbox's root directory, the policy used by other WASI implementations [45], [3], [31]. To verify that the runtime enforces this policy, WaVe tracks each function in the runtime that can access the filesystem paths via the `PathAccessAt` effect. WaVe then verifies filesystem isolation by proving that the runtime cannot cause `PathAccessAt` effects outside the sandbox's root directory. Similarly, WaVe verifies memory safety (§4.1) and network isolation (§4.3) by placing constraints on other effects.

Figure 7 shows the seven different effects that WaVe currently models in order to verify its three safety properties. Some effects correspond directly to system calls (e.g., `SockCreation` corresponds to the `socket` system call), whereas others refer to a type of OS resource being accessed (`PathAccessAt` for an unopened file, and `FdAccess` for an already opened file). Some are even more general: the `MemRead` and `MemWrite` effects are used both to model kernel memory accesses to userspace buffers (e.g., the `read` system call) and potentially unsafe unsafe userspace accesses to memory via functions like `memcpy`, which is used to quickly copy data into and out of the sandbox.

The effects trace WaVe records a list of every effect that could occur during the execution of each hostcall in the *effects trace*. By applying a safety policy to the this trace, WaVe can reason about all possible sequences of effects that can occur during execution. The trace is a *ghost variable*: it exists only at verification-time. Thus, the trace can only be referenced by other verification-time constructs (e.g., function contracts), not by actual code. Finally, the trace is append-only: the history of effects that have occurred cannot be rewritten. This makes it easier to reason about traces at the cost of expressiveness (e.g., we cannot prove that every file a sandbox opens gets closed, §9); in practice, the append-only trace is sufficient for modeling safety properties enforced by existing runtimes.

At verification-time, WaVe inspects and manipulates the trace using three operations:

```
fn len(&self) -> usize;
fn lookup(&self, idx: usize) -> Effect;
fn push(&mut self, effect: Effect) -> ();
```

WaVe uses `len` to inspect the length of the trace to, for example, apply the constraint that after WaVe appends an effect to the trace, `old_trace.len() + 1 == trace.len()`. WaVe uses `lookup` to inspect the effect at a provided index into the array (for example, `trace.lookup(trace.len() - 1)` returns the most recent effect). WaVe uses `push` to append an effect to the trace. WaVe pushes effects to the trace inside the specifications for system calls and unsafe runtime functions.

Specifying effects Specifications for syscalls (and unsafe runtime functions) in WaVe describe how functions alter the trace. They do so using a two-state predicate which relates the structure of the trace upon entering an unsafe function to the trace upon exiting the function. For example, the specification for the `openat` syscall is:

```
#[ensures(effects!(trace,effect!(
    PathAccessAt, dirfd, path, ...
)))]
fn openat(dirfd: usize, path: [u8; 4096],
          flags: i32) -> isize;
```

This `ensures` annotation is a postcondition on `openat` that specifies that the syscall accesses `path` relative to directory `dirfd`; therefore, it extends the trace by the `PathAccessAt` effect, while leaving the rest of the trace unmodified. Each function that invokes this syscall is now responsible for proving that, across all possible executions, the syscall only emits effects that satisfy the runtime's security policy. If, on any possible execution, the runtime does *not* correctly constrain all effects, the verifier produces an error.

Extending WaVe's specifications WaVe proves the safety of the runtime by constraining the kinds of effects that can occur—so WaVe is restricted to proving safety over the effects that it explicitly tracks. Those effects (Figure 7) may not be sufficient to prove *all* safety policies that a runtime may require in the future. For example, if developers are concerned about timing attacks and want to restrict the ways in which a sandbox can access clocks on the host system, the effects in Figure 7 are not sufficient. Since we can't predict every policy that developers may want to prove—and therefore, what effects they need to track—we designed WaVe's effects system to be easily extensible.

To extend WaVe with a new effect and a new policy governing how that effect should be constrained, there are three things a developer must do. First, they must add the effect to the enum shown in Figure 7. Then, they must update the specifications of any syscalls or runtime functions that emit this effect (e.g., for the clock example, specify that `clock_gettime` retrieves the current time). Finally, they need to add checks—guided by the verifier—to satisfy the new safety policy.

The WaVe-hostcall safety contract WaVe ensures that for every hostcall exposed to the sandbox, if the trace was safe when entering the hostcall (with respect to the specified safety policy) then it is safe after the hostcall. Furthermore, WaVe is robustly safe [46]: its safety policy holds regardless of what arguments the sandbox invokes the hostcall with. Since WaVe proves that the trace is safe before and after every hostcall, and the trace is empty (and therefore vacuously safe) on startup, *WaVe proves that for any sequence of hostcalls, with any arguments, the safety policies always holds.*

4. Verification

In this section, we show how WaVe verifies its three safety properties—memory safety, filesystem isolation, and network isolation—and end by explaining how WaVe checks that the sandbox has been set up and torn down correctly.

4.1. Memory safety

WaVe verifies that the runtime and OS do not violate Wasm's memory safety guarantees—namely, that all system calls and unsafe memory operations that the runtime performs (e.g., `memcpy`) read and write only to the sandbox's linear memory. By verifying that every hostcall the sandbox makes is memory safe, WaVe extends the sandbox's memory safety guarantee outside the sandbox itself, ensuring end-to-end memory safety. Proving end-to-end memory safety is a prerequisite for proving resource isolation properties like network isolation (§4.3) and filesystem isolation (§4.2), since these properties rely on the integrity of runtime data structures, which may be compromised if the sandbox can read and write to arbitrary locations in memory.

Next, we introduce the effects WaVe uses to prove memory safety, explain how WaVe constrains those effects, and finally discuss how WaVe verifies the memory safety of more complex system calls, like `readv` and `writev`.

The `MemRead` and `MemWrite` effects To verify that all unsafe memory operations fall within the sandbox's linear memory region, WaVe tracks two effects: `MemRead` and `MemWrite`. `MemRead(addr, len)` and `MemWrite(addr, len)` denote that the runtime or OS is performing an unsafe read/write of `len` bytes starting at `addr`. The specifications for syscalls (e.g., `read`, `write`, or `getrandom`) and unsafe memory operations (e.g., `memcpy`) emit `MemRead` and `MemWrite` effects. For example, the specification for the `read` system call is:

```
#[ensures(effects!(trace,effect!(MemWrite, buf, len)))]
fn read(fd:usize, buf:*mut u8, len:usize) -> isize;
```

This states that `read` system call writes `len` bytes to `buf`.

Proving memory safety WaVe verifies that every `MemRead` and `MemWrite` respects the sandbox's memory isolation policy, i.e., falls within the sandbox's linear memory region. Specifically, for each `MemRead(ptr, len)` and `MemWrite(ptr, len)` effect, WaVe verifies:

```
(1) ptr >= linmem_base &&
(2) ptr + len < linmem_base + linmem_len &&
(3) linmem_base <= linmem_base + linmem_len &&
(4) ptr <= ptr + len
```

Lines 1–2 of this predicate check that the buffer the runtime is writing to or reading from is in the sandbox's linear memory. Lines 3–4 check that neither the buffer nor the linear memory region overflow, i.e., that there are no integer overflows in the bounds checking. WaVe uses this specification to verify both simple I/O operations that use `read` and `write`, which perform a single read/write to memory, and more complex vectored I/O operations with `readv` and `writev`.

Verifying Vectored I/O Vectored I/O allows programs to perform multiple reads to (or writes from) different non-contiguous userspace buffers in a single system call. For example, consider the `readv` system call:

```
readv(int fd, const struct iovec *iov, int iovcnt)
```

This system call reads iovcnt buffers from the file fd into the buffers iov [47]. For a readv call to be safe, WaVe proves that each iovec element in iov, of type:

```
struct iovec {
    void  *iov_base;    /* Starting address */
    size_t iov_len;     /* Number of bytes */
};
```

is within the sandbox's linear memory. To prove this, WaVe appends iovcnt MemWrite effects to the trace, and checks them exactly as it would check the Memwrite created by a single read call. This way, WaVe can apply its single, centralized security specification to many different system calls, which reduces the size of the trusted computing base.

4.2. Filesystem isolation

WaVe allows sandboxes to access the files within a single directory—the sandbox's *root directory* that the runtime specifies at module instantiation-time. This policy is similar to those used in the file system isolation components of Linux namespaces [48] and FreeBSD jails [49], but is finer-grained: instead of applying the jail per-process or per-thread, Wasm runtimes use per-sandbox jails. WaVe enforces this per-sandbox filesystem jail policy by exclusively accessing paths through system calls like openat and mkdirat.

System calls like openat(dirfd, path,...) and mkdirat(dirfd, path,...) access files (here path) relative to the dirfd file descriptor. For example, openat(dirfd, "a.json",...) opens /sbx/a.json if dirfd refers to the /sbx directory. WaVe exclusively uses *at calls for file system access to ensure that each sandbox resolves paths relative to its own root directory, rather than the host's current working directory. When the sandboxed application makes a non-*at system call, WaVe rewrites it to a *at call: open("a.json") becomes openat(sandbox.rootdirfd, "a.json")[1]. By design, all path-based hostcalls in WASI can be rewritten this way [17].

Using *at system calls alone, however, is not sufficient to prevent the sandbox from escaping its root directory: for example, openat(sandbox.rootdirfd,"../secret.txt") will still traverse the sandbox's root directory and access an illegal file. WaVe prevents this illegal sandbox behavior by tracking and constraining the PathAccessAt effect.

The PathAccessAt effect WaVe enforces filesystem isolation by constraining the PathAccessAt(dirfd, path, follows_symlinks) effect. The PathAccessAt effect says that the runtime accessed a path on the filesystem relative to file descriptor dirfd, i.e., a path access through a *at system call, by potentially following symlinks (follows_symlinks). For example, the specification for the open system call is:

```
#[ensures(effects!(trace,effect!(
    PathAccessAt,
    dirfd, path, !flag_set(flags, libc::O_NOFOLLOW)
)))]
pub fn openat(
    dirfd: usize, path: [u8; 4096], flags: i32
    ) -> isize;
```

1. This is done inside the sandbox by a WASI-compatible version of libc [50]

The follows_symlinks argument denotes that the path will expand terminal symlinks. Different system calls have different semantics for expanding symlinks: readlinkat does not follow a symlink if it is the last file in the path (instead, it reads the contents of the symlink), while openat will follow terminal symlinks unless the O_NOFOLLOW flag is set. WaVe expands symlinks before performing safety checks on paths to make the actual safety checking as simple as possible. And, to ensure that the runtime expands symlinks correctly—and can therefore reason about path access control—WaVe verifies its symlink expansion algorithm.

Verifying symbolic link expansion Before checking whether a path is within a sandbox's root directory, WaVe expands all symlinks within the path. This prevents cases in which a sandbox rooted at /sbx accesses a path /sbx/evil/data.json which is seemingly within its root directory—but actually contains a symlink outside the root (e.g., evil → /secrets).

Symlink expansion is a complex recursive procedure that mirrors OS path resolution in userspace—and, unsurprisingly notoriously error-prone [26], [51]. For example, the POSIX path resolution specification [52] states that the OS decides to expand a symlink based on four different factors: (1) what system call is being invoked; (2) whether the current directory entry being processed is the final component in the path; (3) whether particular flags are set (e.g., AT_SYMLINK_NOFOLLOW, AT_SYMLINK_FOLLOW, and O_NOFOLLOW); and (4) how many symlinks have already been expanded.

WaVe safely navigate this complexity by statically verifying *that the expansion procedure does not produce paths that (still) contain symbolic links*. This is necessary and sufficient for reasoning about filesystem isolation. Even if WaVe's symlink expansion algorithm contains bugs, WaVe can guarantee that it doesn't violate filesystem isolation.

Mechanically, WaVe verifies two conditions. First, it verifies that for all prefixes of a resolved path, each prefix does not refer to a symlink:

```
forall(|i: usize| (i < path.len() - 1) ==>
    !is_symlink(path.prefix(i)))
```

The prefix(n) method returns the first n components of the path: for the path ./foo/bar, path.prefix(0) is ./, path.prefix(1) is ./foo, and path.prefix(2) is ./foo/bar. The is_symlink predicate is true for symlinks (as checked by the readlinkat system call). Second, WaVe verifies that the final path entry is either (1) not a symlink, or (2) part of a path WaVe has translated for a system call that does not follow terminal symlinks (e.g., readlink):

```
!follows_symlinks ||
    (follows_symlinks && !is_symlink(&path))
```

By asserting these two conditions, WaVe checks that, for every possible path output by its symlink expansion algorithm, each path component is not a symlink. This is true even when symlinks are recursively expanded: when WaVe finds a symlink in a path, it also guarantees that all path components in that symlink have been properly expanded.

Verifying filesystem isolation Once WaVe has expanded all symbolic links in a path, it can verify that the path is within the sandbox's root directory. For the effect

`PathAccessAt(dirfd, path, follows_symlinks)`, WaVe verifies three conditions:

```
(1) is_relative(path) &&
(2) dirfd == ctx.rootdirfd &&
(3) forall(|i: usize| (i < path.len() - 1) ==>
    (depth(path.prefix(i)) >= 0)
```

The first two conditions—that `path` is relative and `dirfd` is the sandbox's root directory—guarantee that the runtime interprets `path` relative to the sandbox's root [52], [53]. To guarantee that the sandbox stays within its root directory, WaVe statically verifies a third condition: that for all paths, each of its prefix paths have a *path depth* of at least zero. We compute path depth by essentially counting the number of `../` components; for example, `./foo/../..` has depth -1, while `./` has a depth of 0 and `./foo/bar` has a depth of 2. By restricting all path prefixes to have a depth greater or equal to zero, WaVe disallows paths like `../foo`, which have a depth of zero, but still lie outside the sandbox's root directory, and ensures that the sandbox never escapes its root directory.

4.3. Network isolation

WaVe verifies that a sandbox's network access conforms to the WASI sockets proposal [29]. This proposal specifies that sandboxes should be able to make outgoing TCP/UDP over IPv4 network connections, but only to addresses (i.e., ip:port pairs) present in an allow-list that the host application creates at module instantiation-time [2]. This policy can be used to, for example, allow a sandbox to connect to a database server on the local network, but disallow it exfiltrating data to an outside server. By verifying this proposed interface, we provide a verified reference implementation of WASI sockets, which can guide future safe implementations. To prove the WASI-socket safety policy holds for all hostcalls, WaVe tracks two effects: `SockCreation` and `NetConnect`.

The `SockCreation` and `NetConnect` effects To prove that the runtime enforces network isolation, WaVe tracks two effects: `SockCreation(domain, socktype, protocol)`—the sandbox has created a socket of the recorded domain, type, and protocol—and `NetConnect(ip, port)`—the sandbox has connected a socket to an ip:port address. The OS specification for the `socket` system call emits the `SockCreation` effect and the `connect` system call emits the `NetConnect` effect. By constraining these two effects, WaVe verifies that the sandbox can only create sockets of known protocols (TCP and UDP over IPv4), and that the sandbox's sockets can only connect to network addresses in the sandbox's allow-list.

Proving network isolation To prove that every socket the sandbox creates belongs to known protocols, WaVe matches on the `domain` and `socktype` of `SockCreation` effects:

```
domain == IPv4 && (socktype == TCP || socktype == UDP)
```

This lets WaVe immediately reject unknown network protocols that might have surprising semantics, e.g., the sandbox creating a UNIX domain socket and performing IPC.

2. Since the implementation of WaVe, the proposal has been extended with support for domain name resolution and IPv6. We leave the verification of this extended interface for future work.

To prove that the sandbox only connects to addresses in its allow-list, WaVe verifies that the runtime (dynamically) checks that the address is in the allow-list. Specifically, WaVe verifies that for each `SockConnect(ip,port)` effect, the `ip` and `port` are in the allow list:

```
ctx.net_allow_list.contains(addr, port)
```

WaVe uses a fixed-size allow-list (an array of ip:port pairs), so the `net_allow_list.contains` predicate simply checks that the `addr` and `port` arguments are equal to the first entry, or the second, or the third, etc. Moreover, WaVe verifies that the allow-list has not been modified as a part of its `VmContext` well-formedness contract (§4.4).

WaVe verifies that the allow-list has not been modified for two reasons: the first is simply that the WASI-socket API does specify any way for a sandbox to be able to change its own allow-list. The second reason is that, to verify that a socket is within the allow-list at time-of-use, WaVe must know that the allow-list has not been altered since the hostcall checked it. We use a similar technique to prove that, throughout execution, from setup to teardown, the sandbox's allocated memory region is not unsafely modified.

4.4. Setup and teardown

Wasm sandboxes guarantee that they never access memory outside of their allocated memory regions; however, this memory safety guarantee is predicated on the runtime correctly setting up the sandbox by (1) allocating the sandbox's 4GB linear memory region and (2) making sure that accesses beyond the linear memory trap [54]. Existing Wasm runtimes do this in two steps: they use `mmap` to allocate 8GB of read-write memory and then, using `mprotect`, they make the second 4GB a no-access guard page. WaVe statically checks that the runtime correctly sets up this region, and that hostcalls can never invalidate the linear memory region.

To verify that the runtime correctly set up linear memory, WaVe simply checks that the runtime's linear memory pointer is the result of calling `mmap` and `mprotect` with correct arguments. At teardown, WaVe ensures that the runtime's teardown function `munmaps` this region. This specification for setup and teardown is specific to one style of enforcing linear memory isolation (e.g., if the sandbox explicitly checks bounds before every memory access, the guard page could be excluded); however, this guard-page model is what runtimes use in practice [3], [31], [42].

Beyond the setup and teardown, WaVe also verifies that hostcalls never invalidate the sandbox's linear memory region (e.g., by changing page permissions or altering the linear memory pointer) as a part of a well-formedness contract:

`VmContext` well-formedness WaVe maintains a `VmContext` structure per sandbox. The `VmContext` contains metadata about its sandbox, including the location of the sandbox's linear memory, the list of file descriptors the sandbox has opened, and the arguments and environment for the sandbox. WaVe verifies that hostcalls—and the rest of the runtime outside the TCB—are only able to modify the context in safe ways, since the safety of hostcalls depends on the context being well-formed. The well-formedness predicate is shown

```
1 pub fn ctx_well_formed(ctx: &VmCtx) -> bool {
2     ctx.memlen == LINEAR_MEM_SIZE &&
3     ctx.argc < 1024 &&
4     ctx.envc < 1024 &&
5     ctx.arg_buffer.len() < 1024 * 1024 &&
6     ctx.env_buffer.len() < 1024 * 1024 &&
7     netlist_unmodified(&ctx.net_allow_list) &&
8     valid_linmem(raw_ptr(ctx.mem.as_slice()))
9 }
```

Figure 8: WaVe's `VMContext` well-formedness predicate.

in Figure 8. On line 2, it checks that the size of linear memory for the sandbox has not changed since compile-time. On lines 3–6, the predicate sets an upper bound on the number of arguments (and environment variables) and the total size for these arguments. On line 7, it ensures that the network allow-list has not been modified since the host application initialized the sandbox (§4.3). On line 8, the predicate checks the pointer to the linear memory has not been modified.

5. Implementation

WaVe is written in Rust, and verified using the Prusti [30] verification framework. In total, WaVe consists of 7264 lines of code (LoC). The bulk of this code (5907 LoC) is untrusted—both the runtime (4646 LoC) and proof of safety (1261 LoC) are checked.

The remaining code (1357 LoC) is our TCB. The specifications and proof definitions make up 809 lines in the TCB: the safety policy is 43 lines of code, the OS specification (for Linux and MacOS together) is 567 lines of code, and verifier definitions (e.g., defining the effects trace) make up the last 227 lines. The other 548 lines of code in the TCB are our extensions to Prusti. Since Prusti does not support certain features (e.g., bitwise operations) we added trusted wrappers (e.g., for each bitwise operation). As Prusti matures (e.g., support for bitwise operations [55] is currently in progress), we hope to eliminate this code from the TCB.

6. Correctness evaluation

We evaluate the correctness of WaVe's specification and implementation by asking two questions:

1) Does WaVe's specification accurately model real OS and trusted code behavior?
2) How does the semantics of WaVe's WASI implementation compare to those of other runtimes?

To answer the first question, we fuzz the pre- and post-conditions of WaVe's trusted code, including the system call interface. We find no bugs in WaVe's OS specification, but one (non-isolation-breaking) in WaVe's sandbox-teardown. To answer the second question, we create a differential fuzzer for testing runtimes' WASI implementations. We find that WaVe's WASI semantics agree with the majority for all of our fuzzer testcases, but find a number of inconsistencies in other runtimes. We describe these results in detail next.

Checking specifications Our specification fuzzer extracts the specification from trusted functions and uses them to generate test cases for the trusted functions. Specifically, our fuzzer (1) uses specification pre-conditions to generate test cases that are well-formed; (2) runs those test cases on a given trusted function using QuickCheck [56]; (3) checks that the function's output conforms to the expected post-condition—any violation signals a bug in the specification. To check specs for system calls that directly modify linear memory (`readv`, `writev`, `preadv`, and `pwritev`), we use AFL [57] to fuzz a C program that calls each syscall, and ensures that the syscall's memory accesses match our specifications. Before each system call, our tool removes read and write permissions from all mapped pages except the code segment, the top of the stack, the pages required by AFL, and the location the specification says should be written. Then, during execution, if one of these syscalls attempts to access an address that does not have proper read or write permissions, the call will result in a tool-detectable EFAULT error.

After running each fuzzer for 24 hours, we found a single bug: the teardown function was not properly closing the home directory file descriptor. This bug does not break isolation, and therefore falls outside of the scope of WaVe's safety specification, but it could cause a denial-of-service by exhausting the file descriptor-space of the host process. Note also that this type of testing isn't foolproof (e.g., it relies on a testcase hitting a bug, and relies on the kernel to correctly return EFAULTs)—but it does build confidence that our understanding of the syscalls' specifications are correct.

Fuzzing WASI implementations To determine how WaVe's WASI semantics compare to those of other runtimes, we create a differential fuzzer that (1) randomly generates test files that contain sequences of system calls and (2) executes those test files on different runtimes, comparing their behavior. The fuzzer takes as input a list of constraints around inputs to system calls (e.g., it's illegal to close a file that's been closed before). It uses this list to generate a file containing likely-legal long chains of syscalls, and then logs an execution trace of each runtime on the file; this trace includes the return value (if it exists), modifications of pointers passed into the call, and modifications to any environment variables. Any inconsistency in the traces signals a bug or an under specification in WASI, since each runtime intends to implement the same Wasm [54] and WASI [17] standard (§2). Our fuzzer finds differences across four different runtimes—Wasmer [44], Wasmtime [31], Wamr [5], and WaVe. We don't find any bugs in WaVe, but highlight three interesting bugs in other runtimes next.

The first bug is in Wasmer and Wasmtime implementations of `posix_fallocate(fd, offset, length)`. If the size of the file is less than `offset+length`, the file is supposed to increase in size to this sum; otherwise, the file size should be left unchanged. With an `offset` and `length` of zero, however, Wasmer incorrectly truncates the file size to `offset+len` on both Linux and MacOS [58]; Wasmtime [59] incorrectly truncates the file size on MacOS. Wasmtime confirmed the bug [59] and are working on a fix.

The second Wasmer issue [60] arises when a file that has been opened more than once (resulting in multiple different file descriptors) is closed. Closing one of these file descriptors should still allow the other file descriptors to be used: instead, in Wasmer, they are invalidated. For example, after calling `close(fd1)`, a subsequent call to `read(fd2, ...)` fails even if there are still bytes left to be read in the file. Finally, Wasmer does not correctly enforce access modes: it allows writing to file descriptors opened with the `O_RDONLY` flag, and reading from file descriptors opened with the `O_WRONLY` flag [58].

7. Performance evaluation

We evaluate WaVe's runtime performance by answering the following questions:

1) What is WaVe's overhead on individual hostcalls?
2) What is WaVe's overhead on end-to-end applications?

To answer these questions, we measure the performance of WaVe compared to Wasmtime [31], a state-of-the-art industrial WebAssembly runtime. We exclude the other two runtimes from Section 6, WAMR [5] and Wasmer [44], since they did not successfully complete the performance benchmarks (e.g., they failed the SQLite benchmark integrity checks). We evaluate WaVe's performance on three sets of benchmarks: LMbench [32], SQLite [33] (version 3.38.0), and the SPEC2006 CPU benchmarks [34]. We find that WaVe has comparable performance to Wasmtime; using a verifiably secure runtime does not unduly burden performance. In fact, on four of six microbenchmarks and all seven end-to-end applications, WaVe outperforms Wasmtime.

Machine setup We run all experiments on a 2.1GHz Intel Xeon Platinum 8160 machine with 96 cores and 1 TB of RAM running Arch Linux 5.16.4. We compile all benchmarks using the Clang compiler (version 10.0.0) to compile from C/C++ to Wasm, then compile the results to x86-64 using one of two toolchains: Wasmtime (version 0.31.0), or the Wasm2c compiler [61] (as it is interoperable with WaVe). To improve the consistency of all experiments, we isolate benchmarks to a single CPU and disable hyperthreading, dynamic CPU frequency scaling, and Intel TurboBoost. Furthermore, we run all benchmarks on a `ramfs` to improve the consistency of filesystem operations (of the OS itself): since we are primarily interested in runtime overheads (rather than the baseline system call overhead), we aim to reduce noise from the OS as much as possible.

7.1. What is WaVe's overhead on single hostcalls?

To measure WaVe's overhead on individual hostcalls, we use LMbench [32], a benchmark suite designed to measure the latency of OS services (e.g., syscalls, context switches, etc). We use the relevant syscall latency benchmarks (i.e., we exclude syscalls that are not supported by WASI) as a way to measure hostcall latency. To compare WaVe and Wasmtime's performance, we measure six hostcalls, including a null call that measures trampoline overhead of the runtimes by measuring the time it takes to make a hostcall into the runtime and immediately return. We also measure two WaVe hostcalls that are not implemented in Wasmtime: `socket` and `connect`. For each hostcall, we run one million consecutive trials and report the average latency.

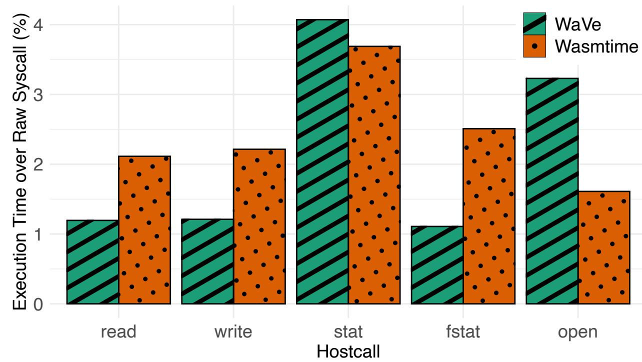

Figure 9: Average hostcall execution time as a percentage compared to raw syscalls

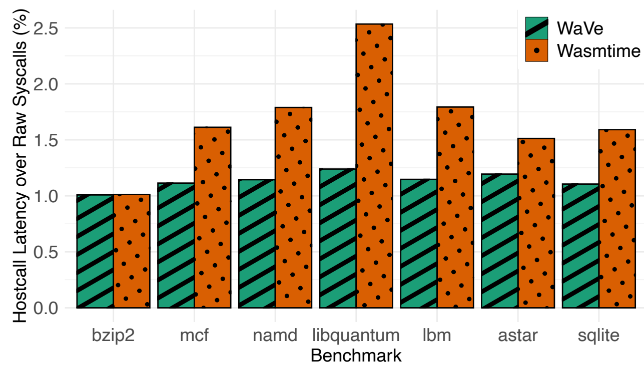

Figure 10: Total hostcall latency for end-to-end benchmarks as a percentage compared to the total hostcall latency with raw syscalls

Figure 9 shows results for the LMbench micro benchmarks. WaVe's null call is 99ns faster than Wasmtime's null call. Out of the five non-null hostcalls, compared to OS syscalls, WaVe has overheads of $1.1\times$ to $4.07\times$ (mean: $2.16\times$), while Wasmtime has overheads of $1.61\times$ to $3.69\times$ (mean: $2.43\times$). On the two network hostcalls WaVe has an overhead of $1.05\times$ and $1.01\times$ respectively; we don't report Wasmtime numbers since Wasmtime does not yet implement this WASI proposal.

In four of the six hostcalls implemented in both WaVe and Wasmtime, WaVe is faster. The remaining two hostcalls, `stat` and `open`, require resolving the symbolic links in a path (see Section 4.2 for details). Wasmtime's path translation is more mature and better optimized—and consequently, it translates paths faster than WaVe. We plan to implement Wasmtime's optimizations in the future, and by verifying both algorithms under the same safety policy, verify that this optimized path translation process is just as safe as our more-naive path translation algorithm. Even with slower path translation, using WaVe today should not meaningfully impact performance: calls that require path resolution are infrequent—there are generally many `read`/`write` calls for each `open`—and thus have little impact on the performance of end-to-end applications as we show next.

7.2. What is WaVe's overhead applications?

We evaluate WaVe on two sets of end-to-end applications: the SPEC 2006 benchmarks [34] and SQLite's speed benchmarks [33]. For both sets of benchmarks, we measure the sum round-trip latency of executing hostcalls—the total time that the benchmark spends in the runtime, from the time that the sandbox invokes a hostcall (e.g., open) to the time that control returns to the sandbox application, including the time spent in the OS. We also measure the total time spent in the OS to act as a baseline latency—by comparing OS latency to total runtime latency, we can deduce the overhead that the runtime incurs. We find that our verified runtime, WaVe, outperforms Wasmtime in all end-to-end applications.

SPEC CPU 2006 SPEC is a popular benchmark suite for measuring the performance of compilers and novel CPU architectures. We evaluate WaVe on the SPEC benchmark not only because it's standard, but also because it exhibits a variety of I/O behaviors: each individual benchmark invokes between 9 and 7,512 hostcalls, with a geometric mean of 488 hostcalls per benchmark. We evaluate WaVe's performance on the six Wasm-compatible SPEC 2006 benchmarks; we only exclude benchmarks that are not compilable with Wasm (e.g., because of exceptions or need for more than 4GB of memory).

The first six columns of Figure 10 show the total hostcall latency for WaVe, Wasmtime, and the OS on SPEC. In all six benchmarks, WaVe outperforms Wasmtime. This is largely incidental: we designed WaVe to be easy to verify using automated reasoning tools, and consequently, the code is simple and has little indirection. This also makes the code easy for the compiler to analyze: we found, for example, that the default Rust release-profile optimizations were able to effectively inline and optimize the hostcall code.

Additionally Wasmtime has some support for WASI asynchronicity, which adds a layer of instrumentation to hostcalls that adds to hostcall latency. How much this layer of indirection contributes to Wasmtime hostcall latency is hard to measure as the code responsible for adding support to asynchronicity is not easily separable from the regular hostcall code. Though both WaVe and Wasmtime can be further optimized, we find these results encouraging: building a high-assurance verified runtime does not need to come at the cost of performance.

SQLite To measure WaVe's performance on end-to-end benchmarks, we evaluate it on SQLite's speed benchmarks. These benchmarks perform common database tasks: two example benchmarks are executing (1) 50,000 INSERTs into table with no index and (2) 10,000 four-ways joins. We use these benchmarks because databases are I/O intensive: SQLite's speed benchmarks invoke 811K hostcalls.

The final column of Figure 10 shows total hostcall latency for WaVe, Wasmtime, and the OS on SQLite speed. WaVe introduces a $1.11\times$ overhead when compared to OS latency, and Wasmtime introduces a $1.59\times$ overhead. WaVe is faster because 800K of the 811K SQLite hostcalls are reads or writes—and WaVe's implementation of these hostcalls are faster Wasmtime's (see Figure 9).

8. Related Work

Our work addresses an open problem—the security of SFI runtime systems [62]. To do so, we draw on a rich history of research into trustworthy software fault isolation and secure system interfaces.

Verified SFI Over the years, there have been numerous efforts to verify the safety of sandboxed code. The most common approach is binary verification [12], [63], [64], [41], [65], i.e., analyzing binaries produced by SFI compilers to ensure that the compiler put all the right safety checks in all the right places. The alternative to binary verification is to formally verify the SFI compiler [66], [67], [13] and ensure that all binaries produced by the compiler are properly sandboxed. While both approaches meet their stated goals of verifying the safety of sandboxed code, they operate under two trust assumptions: (1) that the SFI runtime is fully trusted and bug-free and (2) that all interactions of sandboxed code with the SFI runtime are safe.

Our work compliments these efforts by eliminating the need to trust the runtime or the sandbox interactions with it. Indeed, combining our verified runtime, WaVe, with with a binary verifier like VeriWasm [12], means we can extend Wasm's isolation guarantees end-to-end: VeriWasm ensures that sandboxed code is isolated to the sandbox boundary, WaVe preserves this isolation (and resource isolation) when the sandbox code communicates with the external world.

Modeling and verifying system interfaces Like OS kernels, WaVe exposes a system interface to untrusted code by securely mediating access to the underlying resources. There is a long history of work on modeling and verifying OS kernels, going back to the late seventies work on PSOS [68]. Commuter [69], for example, models how system calls read and write to kernel data structures to find opportunities for conflict-free executions of system calls. Other systems verify safety properties like memory safety and functional correctness of OS kernels [70], [71], [72], [73], [74], hypervisors [75], [76], and TEE safety monitors [77].

A significant amount of effort has been put specifically into verifying the correctness of file system implementations. For example, researchers have built file systems that are functionally correct [78], [79] and secure [80], even in the presence of crashes [81], [82], [83], [84] and concurrency [85], [86], [19]. These developments have even been brought to the world of embedded systems via verified flash file systems [87], [88], [89], and TEEs [90]. Since WaVe relies on operating system correctness, we find this work largely complimentary: by using a verified OS like seL4 [73], for example, we can remove the OS from WaVe's trusted computing base and extend our security guarantees in turn.

Verifying object capabilities Another related line of work is that of object capability verification. Object capabilities can be used to enforce fine-grained privilege separation to protect privileged state from untrusted code [91]. Previous work has developed program logics for reasoning about object capabilities [92] and verified fragments of object capability code in Coq [93], even in the context of sandboxing [94] and OS security [95]. These efforts share a similar goal to WaVe, trying to protect OS state from untrusted sandboxes. WaVe, however, verifies the runtime for a industrial sandboxing interface and reckons with the complex OS semantics needed for doing so.

Hardening system interfaces Multiple systems mediate syscalls to enforce application-level policies when interfacing with the operating system. In the context of high-level language runtimes, for example, several systems isolate native code and restrict this native code's syscall invocations by modifying the underlying runtime (e.g., for Java [96], [97], [98], [99], [100], [101] and .NET [102]). System call interposition, more broadly, is widely deployed—e.g., Linux's `seccomp-bpf` [103] is used to restrict syscalls in systems like Chrome and OpenSSH. Unlike WaVe, these efforts focus on enforcing a single process-level policy (instead of fine grain per-sandbox policies). And, with the exception is Pailoor et al.'s work on using program synthesis to automatically generate syscall policies [104], they lack formal guarantees.

Runtimes for trusted execution environments like Intel SGX [105], [106], [107], [108] share some similarities to WaVe—they must check and translate syscalls' results—but have the added challenge of defensing against an (untrusted) OS from Iago attacks [109] and tricky race conditions [110].

Finding bugs in runtimes While bug-finding does not guarantee safety, finding and removing bugs is a low-cost way to harden a runtime. Previous work has used static analysis to find bugs in the FFI layers of OCaml [111], Python [112], Java [113], [114], and JavaScript [115]. Another popular method for bugfinding in runtimes is fuzzing [116], [117], in particular, for JavaScript engines [118], [119], [120], hypervisors [121], [122], syscall interfaces [123], [19], and file systems [124], [125], [126].

9. Limitations

In this section, we discuss limitations related to both how WaVe verifies and what WaVe verifies—neither the loader nor safety of sandboxes running multiple threads.

Concurrency While WaVe allows for host applications to run multiple sandboxes concurrently (§3.1), it cannot guarantee safety if a single sandbox is running multiple threads concurrently. While this threat model is in line with current Wasm and WASI standards, support for multithreading inside sandboxes is in progress [127], and when it is standardized, developers will expect runtimes to support it. Verifying such a runtime requires reasoning about locking of runtime structures and time-of-check-to-time-of-use bugs, a notorious source of vulnerabilities for secure syscall monitors [26]. We leave this as future work.

WaVe similarly does not reason about concurrent processes. This means, a concurrent process—with the right filesystem permissions—could modify the underlying filesystem to help sandboxed code bypass our filesystem isolation checks [26]. This is possible because our path resolution is not atomic. While we could make resolution atomic (e.g., as Wasmtime recently did), the implications of this more-power attacker model on the rest of WASI are unclear—and, indeed, not considered by other runtimes.

Loader WaVe securely allocates and deallocates linear memory regions for sandboxes, initializes runtime data structures, and handles requests from sandboxes during execution—but it loads the sandbox code from a file to memory with `dlopen`. The dynamic library loading process is complex and error-prone [128], [129], [130]. To provide true end-to-end safety, WaVe would have to verify the dynamic library loading process as well; this is future work.

Expressivity of modular verification WaVe verifies that if a safety policy holds before each hostcall, the safety policy holds after that hostcall. This makes it impossible to express global properties that *aren't* invariant before each call; for example, expressing "all file descriptors opened by the sandbox have been closed properly" *isn't* true before each hostcall, but it may be true during sandbox teardown. We found the current model to be capable of expressing the safety properties for WASI, but, as WASI expands, WaVe may need more sophisticated proof methods.

10. Conclusion

WebAssembly provides safe, portable, and fast isolation—as long as the Wasm compiler and runtime are correct. This paper addresses runtime correctness by verifying that the runtime's sandbox setup and teardown are memory safe, and that the runtime's WebAssembly System Interface implementation preserves memory safety, filesystem isolation, and network isolation. When combined with a binary verifier like VeriWasm or with a verified Wasm compiler (or interpreter), our verified runtime, WaVe, provides an almost complete end-to-end isolation guarantee. These guarantees do not come with the typical cost of verified software: huge proof obligations and poor performance. WaVe is designed to take advantage of modern automated reasoning tools and only requires one (untrusted) annotation for every seventeen lines of code. And, WaVe offers performance similar to production Wasm runtimes like Wasmtime on both microbenchmarks and end-to-end benchmarks like SPEC and SQLite—on most benchmarks, WaVe even outperforms Wasmtime. We did not intend to build a fast runtime. But, it turns out, forcing ourselves to write simple code that can be automatically verified results in code that is easier for the compiler to optimize too.

Acknowledgements

Many thanks to Lin Clark, Chris Fallin, Ranjit Jhala, Tyler McMullen, Till Schneidereit, David Thien, Luke Wagner, and Conrad Watt for fruitful discussions. Thanks to the WAMR, Wasmtime, and Wasmer teams for responding to our reports. This work was supported in part by a Sloan Research Fellowship; by the NSF under Grant Numbers CNS-2155235, CCF-1918573, CNS-2120642, and CAREER CNS-2048262; by a gift from Intel; and by DARPA HARDEN under contract #N66001-22-9-4017.

References

[1] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with WebAssembly," in *PLDI*, 2017.

[2] "WebAssembly," https://developer.mozilla.org/en-US/docs/WebAssembly.

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

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

[5] "WebAssembly micro runtime," https://github.com/bytecodealliance/wasm-micro-runtime, 2021.

[6] B. Dale, "Polkadot's Gavin Wood: WebAssembly is the future of smart contracts, but 'legacy' EVM is right now," 2021. [Online]. Available: https://www.coindesk.com/tech/2021/05/25/polkadots-gavin-wood-webassembly-is-the-future-of-smart-contracts-but-legacy-evm-is-right-now/

[7] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan, "Retrofitting fine grain isolation in the Firefox renderer," in *USENIX Sec*, 2020.

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

[9] "Envoy proxy," https://www.envoyproxy.io/, 2022.

[10] Wasmtime, "cargo fuzz targets for Wasmtime," 2022. [Online]. Available: https://github.com/bytecodealliance/wasmtime/tree/main/fuzz

[11] P. Ventuzelo, "A journey into fuzzing WebAssembly virtual machines," https://i.blackhat.com/USA-22/Wednesday/US-22-Ventuzelo-A-Journey-Into-Fuzzing-WebAssembly-Virtual-Machines.pdf, 2022.

[12] E. Johnson, D. Thien, Y. Alhessi, S. Narayan, F. Brown, S. Lerner, T. McMullen, S. Savage, and D. Stefan, "Доверя́й, но проверя́й: SFI safety for native-compiled Wasm," in *NDSS*. Internet Society, 2021.

[13] J. Bosamiya, W. S. Lim, and B. Parno, "Provably-safe multilingual software sandboxing using WebAssembly," in *USENIX Sec*, Aug. 2022.

[14] C. Watt, "Mechanising and verifying the WebAssembly specification," in *CPP*, 2018.

[15] "Use after free in lucet," https://github.com/advisories/GHSA-hf79-8hjp-rrvq, 2021.

[16] B. Alliance, "Wasmtime security advisories," 2022. [Online]. Available: https://github.com/bytecodealliance/wasmtime/security/advisories

[17] "WebAssembly system interface," https://wasi.dev.

[18] "POSIX.1-2008, IEEE 1003.1-2008," *The Open Group Base Specifications Issue 7*, 2008.

[19] T. Ridge, D. Sheets, T. Tuerk, A. Giugliano, A. Madhavapeddy, and P. Sewell, "SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems," in *SOSP*, 2015.

[20] J. Konka, "Fix rights check for fd_pread and fd_pwrite," https://github.com/bytecodealliance/wasmtime/pull/579, 2019.

[21] P. Hickey, "wasi-common: UNC paths are not handled correctly on windows," https://github.com/bytecodealliance/wasmtime/issues/2650, 2021.

[22] L. Persaud, "Appending to file does not work," https://github.com/wasmerio/wasmer/issues/936, 2019.

[23] whitequark, "Symlink check makes WASI unusable under wine," https://github.com/bytecodealliance/wasmtime/issues/2008, 2020.

[24] ——, "Opening files with O_TRUNC does not truncate them on Windows," https://github.com/bytecodealliance/wasmtime/issues/2009, 2020.

[25] W. Huang, "Add more operand stack overflow checks for fast-interp," https://github.com/bytecodealliance/wasm-micro-runtime/commit/d6e781af281601e6b93601ebfcd6d2fd675960da, 2022.

[26] T. Garfinkel, "Traps and pitfalls: Practical problems in system call interposition based security tools," in *NDSS*, 2003.

[27] Native Client team, "Native Client security contest archive," https://developer.chrome.com/docs/native-client/community/security-contest, 2009.

[28] R. J. Connor, T. McDaniel, J. M. Smith, and M. Schuchard, "PKU pitfalls: Attacks on PKU-based memory isolation systems," in *USENIX Sec*, 2020.

[29] "WASI sockets proposal," https://github.com/WebAssembly/wasi-sockets, 2022.

[30] "prusti-dev," https://github.com/viperproject/prusti-dev.

[31] "Wasmtime," https://wasmtime.dev, 2021.

[32] L. W. McVoy, C. Staelin *et al.*, "lmbench: Portable tools for performance analysis." in *USENIX ATC*, 1996.

[33] M. Owens and G. Allen, *SQLite*. Springer, 2010.

[34] J. L. Henning, "SPEC CPU 2006 benchmark descriptions," in *ACM SIGARCH Computer Architecture News*, 2006.

[35] L. Clark, "Standardizing WASI: A system interface to run WebAssembly outside the web," https://hacks.mozilla.org/2019/03/standardizing-wasi-a-webassembly-system-interface/, 2019.

[36] "Issue 53: SRPC Shared Memory Infoleak / Memory corruption," https://bugs.chromium.org/p/nativeclient/issues/detail?id=53&q=srpc_shm1&can=1.

[37] "CVE-2022-28990 - wasm3 heap overflow," https://nvd.nist.gov/vuln/detail/CVE-2022-28990, 2022.

[38] D. Stefan, "Potential bug in WAMR's `wasm_exec_env_alloc_wasm_frame`," Personal communication with WAMR team, Nov. 2022.

[39] A. d'Antras and D. Stefan, "potential overflow in `get_utf8_string`," Personal communication with the Wasmer security team, Dec. 2021.

[40] D. Stefan, "Fix potential integer overflows in WasmPtr memory access methods," https://github.com/wasmerio/wasmer/pull/2786, 2022.

[41] 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.

[42] "UVWasi," https://github.com/nodejs/uvwasi, 2022.

[43] B. Coenen, "feat(wasi): add rename for a directory + fix remove_dir," 2021. [Online]. Available: https://github.com/wasmerio/wasmer/commit/e0e12f9d9ff41a512e44bd497324eeb0f71abfe2

[44] Wasmer, "Wasmer - universal WebAssembly runtime," https://wasmer.io/, 2019.

[45] Node.js Foundation, "Node.js," https://nodejs.org/en/, 2019.

[46] M. Patrignani and D. Garg, "Robustly Safe Compilation," in *ESOP*, 2019.

[47] "Posix readv man page," https://man7.org/linux/man-pages/man2/readv.2.html.

[48] E. W. Biederman and L. Networx, "Multiple instances of the global Linux namespaces," in *Proceedings of the Linux Symposium*, vol. 1, 2006.

[49] P.-H. Kamp and R. N. Watson, "Jails: Confining the omnipotent root," in *SANE*, 2000.

[50] "WASI libc: a libc for WebAssembly programs built on top of WASI system calls," https://github.com/WebAssembly/wasi-libc.

[51] R. N. Watson, J. Anderson, B. Laurie, and K. Kennaway, "Capsicum: Practical capabilities for UNIX," in *USENIX Sec*, 2010.

[52] "Posix path resolution specification," https://man7.org/linux/man-pages/man7/path_resolution.7.html.

[53] "Posix openat specification," https://man7.org/linux/man-pages/man2/open.2.html.

[54] "WebAssembly core specification," https://www.w3.org/TR/wasm-core-1/.

[55] vakaras, "Prusti bitvectors pr," 2022. [Online]. Available: https://github.com/viperproject/prusti-dev/pull/859

[56] K. Claessen and J. Hughes, "Quickcheck: a lightweight tool for random testing of haskell programs," *ACM SIGPLAN Notices*, vol. 46, no. 4, 2011.

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

[58] Z. Zhao, "Wasmer bugs submitted to security mailing list," 2021.

[59] zzjas, "posix_fallocate truncates file on macos #2973," 2021. [Online]. Available: https://github.com/bytecodealliance/wasmtime/issues/2973

[60] ——, "Double open the same file, close one fd, weird offset issue #3188," 2021. [Online]. Available: https://github.com/wasmerio/wasmer/issues/3188

[61] "wasm2c: Convert wasm files to c source and header," https://github.com/WebAssembly/wabt/tree/main/wasm2c, 2021.

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

[63] L. Zhao, G. Li, B. De Sutter, and J. Regehr, "ARMor: fully verified software fault isolation," in *EMSOFT*, 2011.

[64] G. Morrisett, G. Tan, J. Tassarotti, J.-B. Tristan, and E. Gan, "RockSalt: Better, faster, stronger SFI for the x86," in *PLDI*, 2012.

[65] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC architecture." in *USENIX Sec*, 2006.

[66] F. Besson, S. Blazy, A. Dang, T. Jensen, and P. Wilke, "Compiling sandboxes: Formally verified software fault isolation," in *ESOP*. Springer, Cham, 2019.

[67] J. A. Kroll, G. Stewart, and A. W. Appel, "Portable software fault isolation," in *CSF*, 2014.

[68] R. J. Feiertag and P. G. Neumann, "The foundations of a provably secure operating system (PSOS)," in *MARK*. IEEE, 1979.

[69] A. T. Clements, M. F. Kaashoek, N. Zeldovich, R. T. Morris, and E. Kohler, "The scalable commutativity rule: Designing scalable software for multicore processors," *SOSP*, 2013.

[70] J. Yang and C. Hawblitzel, "Safe to the last instruction: automated verification of a type-safe operating system," in *PLDI*, 2010.

[71] R. Gu, Z. Shao, H. Chen, X. N. Wu, J. Kim, V. Sjöberg, and D. Costanzo, "CertiKOS: An extensible architecture for building certified concurrent OS kernels," in *OSDI*, 2016.

[72] L. Nelson, H. Sigurbjarnarson, K. Zhang, D. Johnson, J. Bornholt, E. Torlak, and X. Wang, "Hyperkernel: Push-button verification of an OS kernel," in *SOSP*, 2017.

[73] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser, "Comprehensive formal verification of an OS microkernel," *TOCS*, vol. 32, no. 1, 2014.

[74] G. C. Hunt and J. R. Larus, "Singularity: Rethinking the software stack," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 2, 2007.

[75] S.-W. Li, X. Li, R. Gu, J. Nieh, and J. Z. Hui, "A Secure and Formally Verified Linux KVM Hypervisor," 2021.

[76] A. Vasudevan, S. Chaki, P. Maniatis, L. Jia, and A. Datta, "überSpark: Enforcing verifiable object abstractions for automated compositional security analysis of a hypervisor," in *USENIX Sec*, 2016.

[77] A. Ferraiuolo, A. Baumann, C. Hawblitzel, and B. Parno, "Komodo: Using verification to disentangle secure-enclave hardware from software," in *SOSP*, 2017.

[78] S. Amani, A. Hixon, Z. Chen, C. Rizkallah, P. Chubb, L. O'Connor, J. Beeren, Y. Nagashima, J. Lim, T. Sewell *et al.*, "Cogent: Verifying high-assurance file system implementations," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 2, 2016.

[79] L. O'Connor, Z. Chen, C. Rizkallah, S. Amani, J. Lim, T. Murray, Y. Nagashima, T. Sewell, and G. Klein, "Refinement through restraint: Bringing down the cost of verification," *ACM SIGPLAN Notices*, vol. 51, no. 9, 2016.

[80] A. Ileri, T. Chajed, A. Chlipala, F. Kaashoek, and N. Zeldovich, "Proving confidentiality in a file system using disksec," in *OSDI*, 2018.

[81] H. Chen, D. Ziegler, T. Chajed, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, "Using crash hoare logic for certifying the FSCQ file system," in *SOSP*, 2015.

[82] T. Chajed, H. Chen, A. Chlipala, M. F. Kaashoek, N. Zeldovich, and D. Ziegler, "Certifying a file system using crash hoare logic: Correctness in the presence of crashes," *CACM*, 2017.

[83] H. Sigurbjarnarson, J. Bornholt, E. Torlak, and X. Wang, "Push-button verification of file systems via crash refinement," in *OSDI*, 2016.

[84] H. Chen, T. Chajed, A. Konradi, S. Wang, A. İleri, A. Chlipala, M. F. Kaashoek, and N. Zeldovich, "Verifying a high-performance crash-safe file system using a tree specification," in *SOSP*, 2017.

[85] T. Chajed, J. Tassarotti, M. Theng, R. Jung, M. F. Kaashoek, and N. Zeldovich, "GoJournal: a verified, concurrent, crash-safe journaling system," in *OSDI*, 2021.

[86] T. Chajed, J. Tassarotti, M. F. Kaashoek, and N. Zeldovich, "Verifying concurrent, crash-safe systems with perennial," in *SOSP*, 2019.

[87] G. Ernst, J. Pfähler, G. Schellhorn, and W. Reif, "Inside a verified flash file system: transactions and garbage collection," in *VSSTE*, 2015.

[88] G. Ernst, "A verified POSIX-compliant flash file system - modular verification technology & crash tolerance," 2017.

[89] G. Schellhorn, G. Ernst, J. Pfähler, D. Haneberg, and W. Reif, "Development of a verified flash file system," in *ABZ*, 2014.

[90] S. Shinde, S. Wang, P. Yuan, A. Hobor, A. Roychoudhury, and P. Saxena, "Besfs: A POSIX filesystem for enclaves with a mechanized safety proof," in *USENIX Sec*, 2020.

[91] M. S. Miller, "Robust composition: Towards a unified approach to access control and concurrency control," Ph.D. dissertation, Johns Hopkins University, Baltimore, Maryland, USA, May 2006.

[92] D. Devriese, L. Birkedal, and F. Piessens, "Reasoning about object capabilities with logical relations and effect parametricity," in *Euro S&P*, 2016.

[93] D. Swasey, D. Garg, and D. Dreyer, "Robust and compositional verification of object capability patterns." *Proceedings of the ACM on Programming Languages*, vol. 1, no. OOPSLA, 2017.

[94] M. Sammler, D. Garg, D. Dreyer, and T. Litak, "The high-level benefits of low-level sandboxing," *Proceedings of the ACM on Programming Languages*, vol. 4, no. POPL, 2019.

[95] G. Heiser and K. Elphinstone, "L4 microkernels: The lessons from 20 years of research and deployment," *ACM Transactions on Computer Systems*, vol. 34, no. 1, Apr. 2016.

[96] J. Siefers, G. Tan, and G. Morrisett, "Robusta: Taming the native beast of the JVM," in *CCS*, 2010.

[97] M. Sun and G. Tan, "JVM-portable sandboxing of Java's native libraries," in *ESORICS*. Springer, 2012.

[98] D. Chisnall, B. Davis, K. Gudka, D. Brazdil, A. Joannou, J. Woodruff, A. T. Markettos, J. E. Maste, R. Norton, S. Son *et al.*, "CHERI JNI: Sinking the Java security model into the C," *ACM SIGARCH Computer Architecture News*, vol. 45, no. 1, 2017.

[99] M. Sun and G. Tan, "Nativeguard: Protecting android applications from third-party native libraries," in *WISEC*, 2014.

[100] E. Athanasopoulos, V. P. Kemerlis, G. Portokalidis, and A. D. Keromytis, "Nacldroid: Native code isolation for android applications," in *ESORICS*. Springer, 2016.

[101] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupé, M. Polino, P. de Geus, C. Kruegel, and G. Vigna, "Going native: Using a large-scale analysis of android apps to create a practical native-code sandboxing policy," in *NDSS*, 2016.

[102] P. Klinkoff, E. Kirda, C. Kruegel, and G. Vigna, "Extending .NET security to unmanaged code," *International Journal of Information Security*, vol. 6, no. 6, 2007.

[103] J. Edge, "A seccomp overview," https://lwn.net/Articles/656307/, Sep. 2015.

[104] S. Pailoor, X. Wang, H. Shacham, and I. Dillig, "Automated policy synthesis for system call sandboxing," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, 2020.

[105] C.-C. Tsai, D. E. Porter, and M. Vij, "Graphene-SGX: A practical library OS for unmodified applications on SGX," in *USENIX ATC*, 2017.

[106] S. Arnautov, B. Trach, F. Gregor, T. Knauth, A. Martin, C. Priebe, J. Lind, D. Muthukumaran, D. O'keeffe, M. L. Stillwell *et al.*, "SCONE: Secure Linux containers with Intel SGX," in *OSDI*, 2016.

[107] C. Priebe, D. Muthukumaran, J. Lind, H. Zhu, S. Cui, V. A. Sartakov, and P. Pietzuch, "SGX-LKL: Securing the host OS interface for trusted execution," *arXiv preprint arXiv:1908.11143*, 2019.

[108] A. Baumann, M. Peinado, and G. Hunt, "Shielding applications from an untrusted cloud with haven," *TOCS*, vol. 33, no. 3, 2015.

[109] S. Checkoway and H. Shacham, "Iago attacks: Why the system call API is a bad untrusted RPC interface," *ACM SIGARCH Computer Architecture News*, vol. 41, no. 1, 2013.

[110] D. R. Ports and T. Garfinkel, "Towards application security on untrusted operating systems." in *HotSec*, 2008.

[111] M. Furr and J. S. Foster, "Checking type safety of foreign function calls," in *PLDI*, 2005.

[112] R. Monat, A. Ouadjaout, and A. Miné, "A multilanguage static analysis of Python programs with native C extensions," in *SAS*, 2021.

[113] G. Tan and J. Croft, "An empirical security study of the native code in the JDK," in *USENIX Sec*, 2008.

[114] G. Kondoh and T. Onodera, "Finding bugs in Java native interface programs," in *ISSTA*, 2008.

[115] F. Brown, S. Narayan, R. S. Wahby, D. Engler, R. Jhala, and D. Stefan, "Finding and preventing bugs in JavaScript bindings," in *IEEE S&P*, 2017.

[116] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, 2019.

[117] X. Zhu, S. Wen, S. Camtepe, and Y. Xiang, "Fuzzing: a survey for roadmap," *CSUR*, 2022.

[118] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *USENIX Sec*, 2012.

[119] H. Han, D. Oh, and S. Cha, "CodeAlchemist: Semantics-aware code generation to find vulnerabilities in JavaScript engines," Jan. 2019.

[120] S. Lee, H. Han, S. K. Cha, and S. Son, "Montage: A neural network language Model-Guided JavaScript engine fuzzer," in *USENIX Sec*, 2020.

[121] S. Schumilo, C. Aschermann, A. Abbasi, S. Wörner, and T. Holz, "HYPER-CUBE: High-dimensional hypervisor fuzzing." in *NDSS*, 2020.

[122] G. Pan, X. Lin, X. Zhang, Y. Jia, S. Ji, C. Wu, X. Ying, J. Wang, and Y. Wu, "V-shuttle: Scalable and semantics-aware hypervisor virtual device fuzzing," in *SIGSAC*. ACM, 2021.

[123] "syzkaller - kernel fuzzer," https://github.com/google/syzkaller.

[124] W. Xu, H. Moon, S. Kashyap, P.-N. Tseng, and T. Kim, "Fuzzing file systems via two-dimensional input space exploration," in *IEEE S&P*, 2019.

[125] S. Kim, M. Xu, S. Kashyap, J. Yoon, W. Xu, and T. Kim, "Finding semantic bugs in file systems with an extensible fuzzing framework," in *SOSP*, 2019.

[126] J. Yang, C. Sar, and D. Engler, "Explode: a lightweight, general system for finding serious storage system errors," in *OSDI*, 2006.

[127] kanishkarj, "WASI multi-threading and atomics," 2021. [Online]. Available: https://github.com/WebAssembly/WASI/issues/296

[128] "Linux kernel ELF core dump privilege elevation," https://isec.pl/en/vulnerabilities/isec-0023-coredump.txt, 2005.

[129] "In the lands of corrupted elves: Breaking ELF software with Melkor fuzzer," https://www.blackhat.com/docs/us-14/materials/arsenal/us-14-Hernandez-Melkor-Slides.pdf, 2014.

[130] "CVE-2017-16997," https://www.cvedetails.com/cve/CVE-2017-16997/, 2017.