Empowering WebAssembly with Thin Kernel Interfaces

Arjun Ramesh
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
arjunr2@andrew.cmu.edu

Tianshu Huang
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
tianshu2@andrew.cmu.edu

Ben L. Titzer
Carnegie Mellon University
Pittsburgh, Pennsylvania, USA
btitzer@andrew.cmu.edu

Anthony Rowe
Carnegie Mellon University, Bosch Research
Pittsburgh, Pennsylvania, USA
agr@andrew.cmu.edu

Abstract

Wasm is gaining popularity outside the Web as a well-specified low-level binary format with ISA portability, low memory footprint and polyglot targetability, enabling efficient in-process sandboxing of untrusted code. Despite these advantages, Wasm adoption for new domains is often hindered by the lack of many standard system interfaces which precludes reusability of existing software and slows ecosystem growth.

This paper proposes *thin kernel interfaces* for Wasm, which directly expose OS userspace syscalls without breaking intra-process sandboxing, enabling a new class of virtualization with Wasm as a universal binary format. By virtualizing the bottom layer of userspace, kernel interfaces enable effortless application ISA portability, compiler backend reusability, and armor programs with Wasm's built-in control flow integrity and arbitrary code execution protection. Furthermore, existing capability-based APIs for Wasm, such as WASI, can be implemented as a Wasm module over kernel interfaces, improving reuse, robustness, and portability through better layering. We present an implementation of this concept for two kernels – Linux and Zephyr – by extending a modern Wasm engine and evaluate our system's performance on a number of sophisticated applications which can run for the first time on Wasm.

CCS Concepts: • **Software and its engineering** → **Virtual machines**; **Operating systems**; **Software safety**; *Runtime environments*; • **Computer systems organization** → Embedded and cyber-physical systems.

Keywords: virtualization, operating systems, WebAssembly, Linux, Zephyr, compilers, interpreters, edge computing

1 Introduction

WebAssembly (Wasm) [52] has emerged as a lightweight, efficient virtualization solution applicable to many domains. As a portable low-level bytecode format with a strict formal specification [12], type system with machine-checked proofs [108], and high-performance implementations [13] with ever-increasing levels of verification [32], Wasm provides an efficient sandboxed execution environment which can run untrusted code at near-native speeds[1]. Primarily deployed in the Web today, it serves as a polyglot compilation target powering many applications such as Photoshop [80], Unity [7], and high performance libraries [2].

Following its success in browsers, Wasm has also gained broad adoption in cloud and edge contexts [4, 79, 102]. To operate in these contexts, Wasm requires a *system interface*, as its core specification defines a portable bytecode ISA without any system interfaces. Outside Web APIs, the WebAssembly System Interface (WASI) [27] is the only proposed standardized platform interface. WASI is a secure, cross-platform (OS-agnostic) Wasm interface specification that couples a system interface with a new capability-based security model, enforcing filesystem isolation with pre-opened directories, network isolation with constrained sockets, and explicitly-enumerated environment variables.

Recent years have seen growing interest in extending Wasm beyond controlled cloud environments to modern cyber-physical deployments at the edge, incorporating highly-capable mobile and internet-connected embedded devices such as IoT [69–71, 78], automotive [40, 90], and industrial

---

[1]The Wasm language specification is continually advancing, incorporating various enhancements for SIMD vectorization [11, 21], larger address spaces (memory64 [20]), and finer-grained memory control (multi memory [14] and custom page sizes [18]).

systems [112]. Unlike the cloud, these domains have several uniquely challenging requirements. Software in these domains often demands high performance and memory efficiency, operates in safety-critical physical environments, and combines components distributed by many vendors in their preferred choice of languages. Applications are also often deployed across a wide gamut of system configurations for long deployment periods. Many critical software stacks are hence presently frozen in time on legacy hardware and software ecosystems that are rapidly falling behind the state of the art for safety, efficiency, and performance.

Wasm is a compelling solution for these problems, addressing efficiency, safety, and polyglot concerns. However, the additional requirements of high portability across software platforms, long deployments, and critical legacy software are system interface concerns which Wasm explicitly does not address. To date, no existing standard system interface for Wasm adequately addresses these challenges. In particular, WASI's goals are somewhat *misaligned* with these challenges:

(1) As a *new* portable API across many operating systems, it must be reimplemented many times;

(2) Its design exploration and evolution make it unstable and therefore unsuitable for long deployments; and

(3) Its divergence from longstanding standards like POSIX means it cannot run existing software.

Furthermore, despite many years of standardization efforts, WASI remains an extremely simplified OS interface, lacking support for prevalent OS features like memory-mapping, processes, asynchronous I/O, and signals. Many applications require these features and simply cannot run on WASI (see Table. 1). Given the many benefits of Wasm as an execution format, we believe the lack of an effective system interface is a key limiting factor to Wasm adoption in many domains.

In this paper, we propose *thin kernel interfaces* for Wasm to enable a new virtualization solution for userspace applications. Our key insight is that operating systems' userspace syscall interfaces are a stable, de facto standard upon which thousands of desktop, server, and embedded applications have been and continue to be built. By creating a thin Wasm interface for existing operating systems, we can easily virtualize entire software stacks from the bottom-up with little modification and run them on a diverse set of host ISAs.

In contrast to existing system interfaces for Wasm, thin kernel interfaces do not define an entirely new API surface against which applications need to be refactored, but rather faithfully model the existing underlying operating system. The bottom-up approach means that porting applications only requires recompilation with a Wasm-enabled toolchain, which can be done quite trivially as standard libraries and ABIs are already written against the OS syscall API. Furthermore, kernel interfaces are both complementary and advantageous to higher-level APIs like WASI as a complete syscall
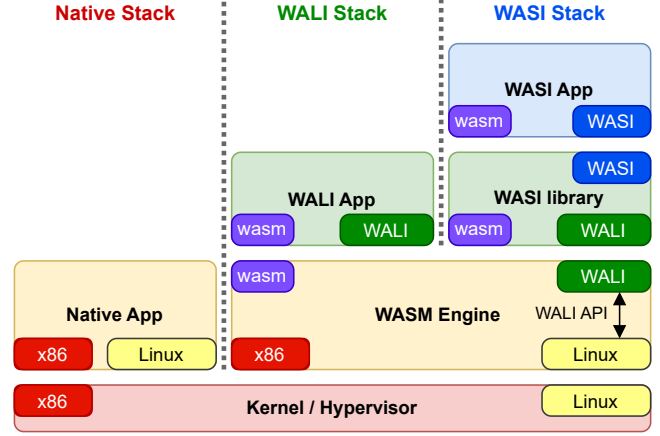

Figure 1. Linux Virtualization stack with WALI as a foundation.

specification allows them to be implemented as individually-sandboxed layers over the kernel interface (Fig. 1) as Wasm modules. Such layering makes high-level Wasm API implementations more **portable**, **safe**, and **reusable**, allowing them to work on any Wasm engine that exposes the same kernel interface. Furthermore, the complex trusted computed base (TCB) of the Wasm engine is now greatly simplified, as high-level APIs once necessarily embedded within the engine are now *decoupled* from it.

We demonstrate the feasibility of this concept with two distinct OS interfaces — a *Linux* interface **WALI**, and a *Zephyr* RTOS interface **WAZI**. Beyond these two implementations, we also present a ***recipe*** for Wasm kernel interface design applicable to any kernel, which addresses important architectural decisions in safely supporting common OS features like signals, processes, threads, and memory mapping to bridge the mismatch between Wasm and a typical OS execution model.

1.1 Motivation

Virtualization Requirements at the Edge Modern edge systems are complex pieces of software deployed on highly-customized hardware platforms. Software virtualization in this setting has the following requirements:

(1) **High ecosystem portability**: Applications in manufacturing equipment or automotive systems are deployed across a wide array of systems and engines. A simple interface implementation must be able to easily port complex software stacks across many engines.

(2) **Long deployments**: Many applications are deployed for decades without modification. Supporting such deployments requires a target with a *stable* set of features that is complete with low churn.

(3) **Critical legacy software stacks**: Many industries have large actively-deployed legacy codebases that cannot easily be rewritten. Applications should be virtualizable

"as-is", with simple update mechanisms allowing for incremental software improvements.

(4) **Efficiency**: Efficient CPU/memory usage and package size is critical in resource-constrained devices deployed in the wild. Highly reactive environments may also depend on fast application startup times.

(5) **Safety and Security**: Safety-critical physical environments such as control systems and factories demand safe execution and resource control. Applications must be statically-typed, verifiable, and isolated from other co-located applications.

(6) **Polyglot**: Applications should be easy to develop and port from a variety of programming languages and support the vast set of software libraries currently available to these languages.

Why Wasm? Given the stakes, safety should be paramount for a virtualization solution, especially in actively-deployed systems susceptible to remote code injection attacks. Containerization (e.g. Docker [8]) is too memory inefficient, and is still vulnerable to control-flow redirection and remote code execution (RCE) attacks. Bytecode VMs often meet the aforementioned virtualization requirements, but targeting most managed runtimes (e.g. Java and CLR) restricts application development to specific families of languages, often with builtin non-determinism and garbage-collection, making them unsuitable in embedded and real-time contexts.

In contrast to the above solutions, Wasm offers a secure and efficient compilation target for numerous high-level languages. Armed with kernel interfaces, Wasm binaries inherently provide the following core properties by design:

(1) **Type Safety**: Wasm binaries are type-safe and statically validated prior to execution;

(2) **Memory Safety**: All Wasm memory accesses, including NULL and type unsafe pointer indirection, are in-memory sandboxed and runtime-enforced. Default memory initialization also prevents undefined behavior;

(3) **Non-Addressable Execution State**: Wasm's static code segments and its dynamic call/value stack are neither directly addressable nor aliasable from within the module, ruling out arbitrary code injection/execution attacks;

(4) **Control Flow Integrity (CFI)**: Wasm's structured control flow paradigm along with typed function calls and a virtualized call stack provides implicit CFI [23].

(5) **ISA Portability**: The Wasm bytecode format and semantics are defined independently of hardware ISAs, allowing universal compatibility with any underlying host;

Protecting System Software Real-world edge deployments often include a large exploitable attack surface of not just applications but dozens or even hundreds of susceptible background system services (e.g. remote logins, authentication,

daemons) and libraries. By using Wasm to secure system daemons (see Table. 1), thin kernel interfaces could have immediately thwarted numerous *OpenSSH* [15] vulnerability exploits, including the critical *regreSSHion* bug [16] that affected millions of Linux devices worldwide. Securing a large swath of the system software stack with Wasm in this manner, however, requires an ideal interface that supports a diverse set of OS features.

The Need for Complete Wasm OS Interfaces Wasm as a virtualization solution for edge systems is clearly promising, but high portability, long deployments, and legacy software support require the right system interface compatible with these goals. Standardized efforts towards this end include:

- **WASI** [27]: a W3C standardization effort that aims to design a completely portable Wasm API with a strict capability security model, fine-grained access controls, a simplified filesystem, and socket virtualization.
- **WASIX** [28]: a rogue superset of WASI, proposed by the Wasmer [9] team that adds missing POSIX functionality to jumpstart WASI development.

As API specifications intent on integrating both OS portability and a capability-based security model, both WASI and WASIX face major design difficulties in maintaining compatibility with existing application APIs like POSIX. For one, essential features like signals, memory-mapping, or users/groups have been eschewed due to design difficulty; as a result, applications that use these *just don't work*. Yet despite WASI's limited feature set, implementing WASI is surprisingly complex[2] with concomitant engine requirements, and many implementations are riddled with bugs [58]. Further exacerbating the problem, WASI is growing with new features for machine learning, HTTP, key-value stores, etc., as all engines engine are forced to implement these internally, in the trusted computing base. Bugs in these implementations can compromise the inherent memory sandboxing, CFI, and RCE defense, effectively breaking Wasm.

Our approach instead adopts a layering approach to API design, targeting historically stable OS-specific syscall interfaces, and hence *decoupling the security model from the feature-completeness* of Wasm as an ISA. By deprioritizing OS-portability as a primary goal, kernel interface implementations remain remarkably tiny (WALI ≈ 2000 LoC) compared to their higher-level API counterparts, and offer feature-completeness for running complex legacy applications, including layered WASI implementations over it. These kernel interfaces implementations are also easily adaptable to support the handful of popular OSes by following our *recipe* in Sec. 5. Engines can now support a plethora of arbitrary high-level APIs for an OS in a portable, sandboxed manner above a single implementation of a stable syscall interface.

---

[2]libuvwasi [5] for preview1 is over 6,000 lines of engine code, excluding *libuv* itself or the component model which is several thousand more!

This is especially valuable for kernels such as Zephyr that to date have *no* engine with a complete WASI implementation.

## 1.2 Positioning Kernel Interfaces in the Ecosystem

Kernel interfaces hold a unique position in the Wasm ecosystem without diminishing the use-cases of existing capability-based security APIs. They enable a new ecosystem providing *traditionally native* binary software stacks (e.g. managed edge systems, OS packages, consumer mobile/desktop applications, and WASI implementations) with both a viable virtualization target for safety and portability, and an opportunity to run alternative security models. Allowing a full set of OS system calls (e.g. Linux) raises some obvious questions:

Q: Do Kernel Interfaces Break Wasm? No, Wasm actively facilitates support for arbitrary custom system interfaces via import sections in its language standards. This paper demonstrates how kernel interfaces can be exposed safely while preserving the most essential Wasm properties listed in Sec. 1.1. Importantly, kernel interfaces do not directly tamper with the execution stack and instructions in WALI/WAZI modules still operate within their own isolated memories (Sec. 3.6). So-called "risky" features like setjmp/longjmp in C/C++ would be compiled to safe, non-local control flow with exception-handling akin to the browser, making them a toolchain concern rather than a system interface issue.

Q: Do Kernel Interfaces Ruin WASI/Browser Ecosystems? No, kernel interfaces are *not meant to replace WASI* or be directly exposed to untrusted cloud software or as browser APIs. WASI and browser ecosystems will continue to operate with their capability-based security models. In fact, we propose no changes to their APIs or security models, which are undoubtedly beneficial for their intended cloud or Web deployment scenarios. We contend however that engines will be more secure if they move their WASI implementations up to Wasm and layering it over kernel interfaces (Sec. 4.1) *without exposing the latter directly* to user applications.

1.3 Contributions

(1) We identify *thin kernel interfaces* for Wasm as the key layering mechanism that simultaneously decouples high-level APIs (e.g. WASI) and engine evolution, improves safety by sandboxing their implementations, and allows numerous legacy applications to run for the first time on Wasm across multiple hardware ISAs.

(2) We design and implement WALI, an interface for Linux, where we support a critical mass of syscalls, explore several process and signal models, and evaluate the implementation on sophisticated real-world applications.

(3) We propose a *recipe* for defining ISA-agnostic virtualization for OS kernels using Wasm and apply this recipe to design WAZI, an interface for Zephyr RTOS.
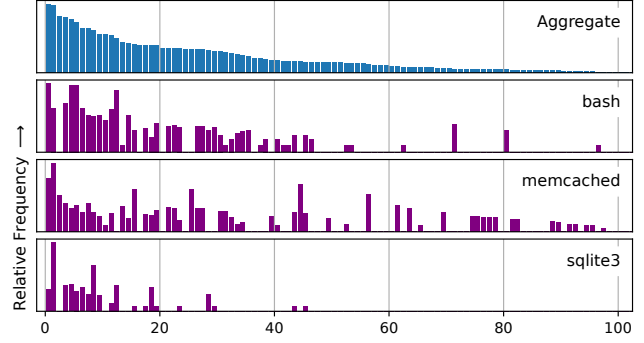
Figure 2. Log-normalized Linux syscall profile sorted by aggregate frequency; the top row shows the distribution of *all invoked syscalls across all benchmarks* sorted by frequency; lower rows show the syscall frequency for each benchmark using the same ordering.
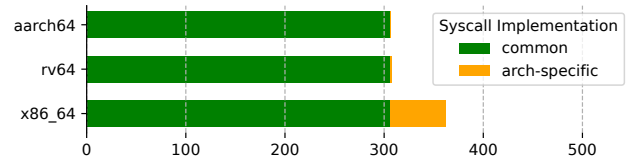
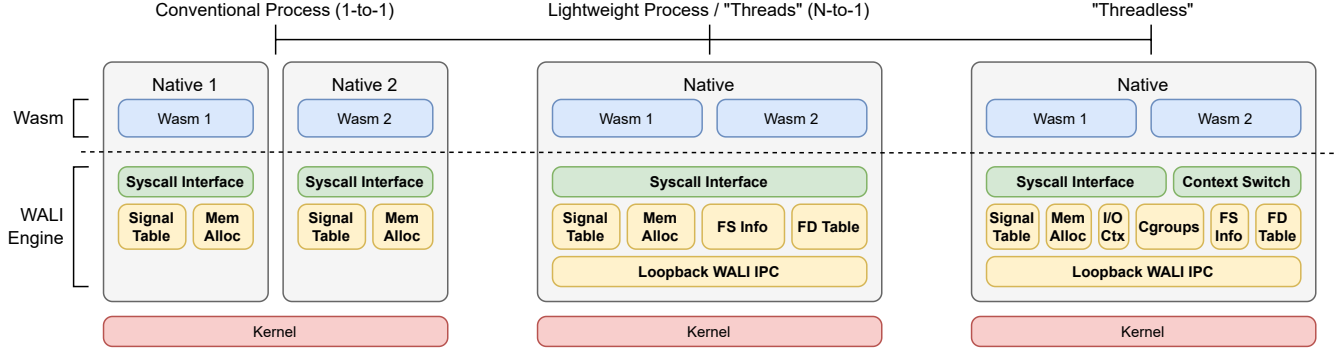**Figure 3.** Similarity of Linux Syscalls across ISAs.

(4) We showcase decoupling WASI from engines by compiling libuvwasi, a popular implementation of WASI, unmodified over WALI with full feature-completeness. The WALI and WAZI implementations are fully open sourced and available at https://github.com/arjunr2/WALI.

2 Scoping Existing System Call Interfaces

Operating systems feature a daunting array of hundreds of system calls, sometimes slightly different across ISAs, including some platform-specific calls. While virtualizing every call across all platforms seems like a gargantuan task, our study of real-world applications found Linux syscalls have enough commonality in *actual use* that WALI can cover the vast majority of OS functionality in a lightweight, portable way.

Practical Prevalence of Syscalls We first evaluated the feasibility of syscall-based virtualization to scope our work on WALI/WAZI by studying the number and frequency of syscalls across a variety of applications. Linux API usage literature reports ≈ 200 (out of 300-400) required syscalls as the critical point with static analysis [101]; more recent work with dynamic analysis [66] however reports a mere 148 syscalls which is consistent with our findings (Fig. 2). This is consistent with our findings (Fig. 2) where many applications use fewer than 100 unique syscalls, and the union of all applications is around 140-150 syscalls. Similarly, Zephyr has ≈ 500 syscalls, but many of them target domain-specific subsystems (GNSS, SiP SVC, Auxdisplay, etc.) or toolchains.

Conventional Process (1-to-1) | Lightweight Process / "Threads" (N-to-1) | "Threadless"

Wasm
Native 1 — Wasm 1
Native 2 — Wasm 2
Native — Wasm 1, Wasm 2
Native — Wasm 1, Wasm 2

WALI Engine

Native 1: Syscall Interface; Signal Table; Mem Alloc
Native 2: Syscall Interface; Signal Table; Mem Alloc

Native: Syscall Interface; Signal Table; Mem Alloc; FS Info; FD Table; Loopback WALI IPC

Native: Syscall Interface; Context Switch; Signal Table; Mem Alloc; I/O Ctx; Cgroups; FS Info; FD Table; Loopback WALI IPC

Kernel | Kernel | Kernel

Figure 4. Process Model Spectrum for varying configurations of *Native* and *Wasm* processes; WALI must implement the **bolded** components.

Thus, WALI/WAZI only need to support a fraction of the total system call interface to run most applications faithfully. We also identify that this subset of syscalls encompasses the core features offered by an operating system – we postulate that most remaining syscalls can easily be auto-generated as kernel *passthrough* methods in the future (see Sec. 5, 6).

Diversity Across ISAs Linux officially supports ≈ 500 syscalls but not all syscalls are available on all ISAs [59]. We found there is a large common core, as both Arm and RISC-V are nearly identical and largely a subset of x86-64 with a handful of differences (Fig. 3). These syscalls persist for backward compatibility but can often be emulated using newer, more secure alternatives (e.g., `access` with `faccessat`, all `stat` variants with `newfstatat`). Meanwhile, Zephyr is conveniently designed to be ISA-portable across all its diverse target platforms/boards. Thus, WALI/WAZI can feasibly attain interface-level ISA portability will minimal effort.

3 An Interface for Linux: WALI

As a ubiquitous, robust, powerful, and stable OS, we choose Linux as the primary target to demonstrate the feasibility of thin kernel interfaces. WALI specifies a set of approximately 150 WebAssembly host functions that can be imported into a Wasm module. Since most Wasm runtimes support extensible host functions, WALI implementations are relatively easy to add to engines.

WALI is comprised primarily of syscalls, with a small number of support methods for environment/command-line parameters. WALI syscalls correspond nearly 1-to-1 with native Linux syscalls and need only translate data between the virtualized syscall interface and the native Linux syscall interface and vice versa. By design, most WALI calls are "*passthrough*", with low-overhead *zero-copy* operations with appropriate translation between the Wasm and Linux memory spaces. WALI maintains fundamental Wasm guarantees (ISA portability, memory sandboxing, CFI, Harvard architecture) and thus disallows register accesses, stack access, and non-local gotos (`setjmp`/`longjmp`).

Considering all these components, WALI must make fundamental important design decisions to bridge the Linux and WebAssembly execution environments — primarily for the process/thread model, memory management model, signal handling, security model, and cross-platform support.

3.1 Process and Thread Model

While most operating systems support concurrency using a process/thread model, core Wasm provides no notion of processes or threads. This requires WALI to present a process/thread model to Wasm applications that faithfully represents the behavior of native Linux processes in order to seamlessly interact with each other and with native processes. Conventional Linux processes commonly interact with each other using pipes, shared memory, or signals, while threads/light-weight-processes (LWP) often share a common memory space and communicate using synchronized memory operations. Many syscalls additionally use *process ids* (PIDs) to target processes for calls that involve signals, usage statistics, scheduling characteristics, and status updates. To faithfully replicate these behaviors, we explore three models for WALI along the spectrum of possible models (Fig. 4):

1-to-1 model In this (simplest) model, each WALI process is assigned a unique native Linux process with its own PID. A key advantage of this model is the ease of implementation and verification: most process/thread-oriented WALI syscalls, including `fork`, can be implemented as pass-through syscalls directly to the kernel. This design also relieves the engine from maintaining any WALI native process/thread state, but places the engine at the mercy of the Linux kernel for further optimizations in performance or inter-process communication. To work around Wasm's lack of a thread model, both the web and WASI support threads via replicating Wasm module instances. This "instance-per-thread" model preserves distinct execution state, including separate value and call stacks[3], globals, and tables. We adopt the *1-to-1*

---

[3]The design around Wasm thread support may be standardized in future with the Shared-Everything Threads proposal [22]

design with an instance-per-thread for our WALI implementation for simplicity, and it incurs zero bookkeeping memory overhead of processes/threads within the engine.

N-to-1 model The *N-to-1* model runs multiple WALI processes as *LWPs* within a single Linux process. *Thread-based* LWPs require virtualization of all unshared native process state, significantly increasing implementation complexity. Luckily, the native Linux clone call supports a precise specification for fine-grained resource sharing with the child process, allowing WALI implementations to optimize trade-offs on the spectrum between conventional "processes" and "threads", e.g.:

- Setting CLONE_VM allows the child LWPs to share the parent's virtual address space, enabling potential memory usage optimizations.
- Disabling CLONE_THREAD makes interactions with virtual LWPs identical to conventional processes: they obtain a unique thread-group ID (TGID) and possess their own scheduling properties.

Co-locating multiple WALI processes in one native process also allows sharing filesystem information and signals, which can allow fast inter-process communication without syscalls which may even outperform native Linux IPC.

"Threadless" model Mode switch overheads for kernel calls are becoming more significant with recent exponential improvements in CPU and memory performance [117]. Leveraging Wasm's sandboxing, further reduction in overhead could be achieved by avoiding an LWP-backed process model in favor of a hyper-optimized process model that runs Ring-0 delegated tasks in user-space [110]. We imagine that a *threadless* model could support context switches as fast inter-instance function calls within the Wasm engine in user-space, eliminating mode switch overheads. TGID-based process identification can be emulated with a *dummy* native process that forwards process-based interaction to the WALI engine, or with basic kernel support for providing raw TGID identifiers.

3.2 Memory Model

Wasm module memory is a 32-bit [4] byte-addressable, bounds-checked linear address-space instantiated as subset of the host process's memory space. Module memory declarations statically specify an initial and maximum number of (64KiB [5]) pages that are shareable by multiple parallel computations (i.e, *threads* on WASI and WALI), and accessible by the Wasm program using load/store instructions that generate a 33-bit index into memory. These memory model details necessitate design choices to support WALI syscalls that operate on data in memory or perform memory management. We transparently support all memory-oriented syscall operations with the following techniques:

Address-Space Translation For many native syscalls that accept arguments representing pointers to process memory regions, WebAssembly memory "pointers" cannot be directly forwarded to native syscalls as pointer types. For such WALI syscalls, the engine must perform an *address-space translation* of memory references between Wasm and native process memory. This fast linear translation with minimal bounds-checking overhead allows most WALI syscalls to be *zero-copy*, enabling high-performance I/O directly from the sandboxed intra-process Wasm memory.

Layout (ABI) Conversion Some native syscalls accept pointers to complex structured-typed arguments, whose byte-level layout and size of may vary across ISAs, making it impossible for WebAssembly to provide these as platform-independent zero-copy syscalls. In such situations, WALI must explicitly perform Wasm-to-native struct copies for input arguments and native-to-Wasm copies for output arguments. Few syscalls (<10%) use such arguments and their sizes are usually small and fixed, imposing minimal overhead.

Memory Management WALI allows nearly all use cases of mmap, mremap, and munmap, including mapping files and other resources with unconstrained address ranges. All allocations are fully sandboxed and mapped by the WALI implementation within the Wasm memory address space. Our implementation automatically grows Wasm memory for new mappings, up to the memory declaration's self-imposed limit, failing if the size grows beyond the maximum. [6] Subsequent unmapping with munmap is performed as a passthrough native syscall with normal bounds-checking.

Prior to WALI, source-level language memory management libraries (e.g. malloc or garbage collection) were reliant on the memory.grow instruction, which required non-trivial porting effort to run on Wasm. With WALI's faithful memory-mapping support, sophisticated mapping strategies work unmodified using kernel interfaces, allowing these libraries to immediately target Wasm. Such support obligates the WALI engine to internally manage the state for allocated mapped and free memory segments. Our implementation allows mapping a region in the engine at most once (which only requires a single bookkeeping variable) for tracking the base address of the allocation pool. Future implementations however may avoid fragmentation with more elaborate allocators – these can implemented as Wasm modules over WALI to reduce engine complexity and improve portability.

---

[4]With the recent standardization of the memory64 proposal [20], Wasm memories can be optionally extended to 64 bits.

[5]A recent Wasm proposal would allow custom power-of-two page sizes [18].

[6]We utilize the MAP_FIXED/MREMAP_FIXED flag to native mmap/mremap syscall to map pages at specific addresses in Wasm memory.
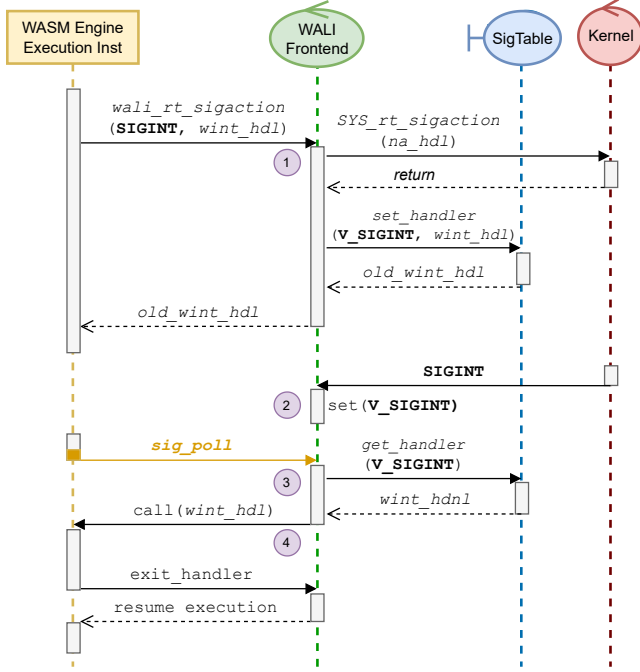
Figure 5. WALI Asynchronous Signal Handling Sequence Diagram

3.3 Signal Model

Both synchronous and asynchronous signal handling are critical features used by many Linux programs. Synchronous signals are generated and delivered immediately to processes in reaction to most hardware faults, e.g. memory access faults, illegal instructions, or arithmetic exceptions. These are easy to catch and trigger *traps* in the Wasm engine for safe exception handling (e.g. SIGFPE for integer division-by-zero). Asynchronous signals however are more challenging: they may be generated and delivered at any point in a process's lifespan, even while the target process is suspended, and are frequently used for software interrupts, job control, termination, or I/O. WebAssembly, at the time of writing, has no standardized instructions for asynchronous callback operations. As a result, the WALI engine must explicitly support both delivering asynchronous native signals at Wasm bytecode *safepoints* [43] and executing user signal handlers faithfully.

Asynchronous Signal Handling WALI engines must be capable of supporting asynchronous signal delivery, masking, and execution of application Wasm functions that handle signals, similar to that of native processes. To fully support asynchronous signal handling, the Wasm engine must support reentrancy where a host function calls back into the same Wasm module from which it was invoked. WALI implementations leverage this capability to effectively virtualize the main stages in a Linux signal's lifecycle: signal registration, generation, delivery, and handler execution (Fig. 5).

(1) Signal Registration: Wasm modules must be able to configure asynchronous signal callback functions, akin to native process, with rt_sigaction. To support this, the WALI engine internally maintains a virtual *sigtable* of registered signals, mapping every Linux signal to a target callback function in the Wasm module. When Wasm modules invoke their virtual wali_rt_sigaction syscall, two things occur:

- The Wasm function pointer (index into a Wasm table) is dereferenced and registered in the *sigtable*; and
- The native rt_sigaction is called within the engine to register a *native handler* for the signal that performs virtual signal generation.

The Wasm function pointer is also saved in the *sigtable* to return back the old action (old_wint_hdl) to the module for future invocations of wali_rt_sigaction. The virtual *sigtable* incurs a minimal bookkeeping overhead of $< 1kB$.

(2) Generation: The engine stores a bit-vector and a queue of pending signals per WALI process to serve as the virtual signal generation mechanism. Since we use rt_sigaction, native signal generation is performed by the underlying kernel which WALI, as a user-space interface, uses to set the signal's bit-vector element and add it to the pending queue.

(3) Delivery: Generated signals remain pending and are delivered shortly after to the native process. However, signals may be blocked using a *signal mask* (with rt_sigprocmask) to prevent delivery until explicitly unblocked. WALI supports virtual signal blocking by maintaining a signal mask per WALI process. Since signal masks are recorded for each *thread* and initial masks are inherited from the parent thread, for any process-model that uses the underlying clone syscall, WALI can just use the Linux LWP's signal mask. Delivered signals are then picked up during execution by any WALI thread in the thread group as a result of native Linux's process model.

(4) Handler Execution: Finally, the WALI engine must trigger the execution of the registered virtual signal handler in Wasm post-delivery. Since asynchronous signals do not impede execution, WALI engines can choose to delay the signal delivery and handling to a later time. However, arbitrary invocations of signal handlers during critical sections in the engine that modify module instance state (memory, tables, globals), execution environment state during call/return instructions, or internal WALI state can break consistency guarantees of the WebAssembly execution model. Therefore, WALI implementations must deliver signals at *safepoints* (sig_poll in Fig. 5), inserted by the compiler, where the state consistency is preserved. WebAssembly instruction boundaries are a natural location for safepoints, but frequent polling for signals impacts performance and compiler optimization opportunities. Given reactivity is often

non-critical, polling at loop headers and/or function entry-points are shown to be effective solutions [30, 55] – our implementation performs the former.

The WALI engine must also be careful to avoid violations to basic signal delivery guarantees. Pending virtual signals blocked by `rt_sigprocmask` must not be delivered until the same is unblocked. This is avoided with an additional safepoint immediately after the native `rt_sigprocmask` invocation within the engine, which handles outstanding generated signals before entering the Wasm critical section. Additionally, when `SA_NODEFER` flag is unset and two identical pending signals occur, a stack-based structure containing signal state can be used to defer new handler execution until the current handler execution completes. Reserved handler types like `SIG_IGN`, `SIG_DFL`, and `SIG_ERR` require specialized support. The engine can allow these to bypass virtual signal handling entirely with direct calls to the kernel and special trap handlers to provide safe handling.

3.4 External Parameters

The WALI specification includes methods for supporting external host parameters like command-line arguments and environment variables within the application sandbox.

Command-line Arguments WALI transparently supports transfer of command-line arguments from the host to the application. To minimize state and increase safety in the engine, WALI delegates the ownership of these variables to the standard library. On startup, the standard library allocates an appropriately sized argument vector using two API methods – `get_argc` and `get_argv_len`. Safe copying of each argument into the WALI process is performed post-allocation using a `copy_argv` method. As a result, any security vulnerabilities exposed through buffer overflows during parsing remain entirely contained within the sandbox.

Environment Variables Initialization of environment variables works similarly to command-line arguments in WALI, where values are not inherited from the parent shell for security reasons but rather explicitly specified when invoking the engine. However, a subtle edge-case arises when executing programs internally invoke `execve`, which must pass virtual environment variables to the child WALI process as opposed to the host engine. One solution for engines is to forward the current virtual environment as command-line arguments when invoking a WALI binary. An alternative elegant engine-agnostic technique we adopt is to use a unique *shared-memory segment* encoded with the WALI process ID to store the virtual environment state before invocation of `execve`, which is picked up by the child process on startup.

3.5 Cross-Platform Support

Architecture-agnostic packaging of Wasm binaries is of prime importance for WALI. Syscalls, however, are often non-portable and vary across architectures both in their syscall numbers
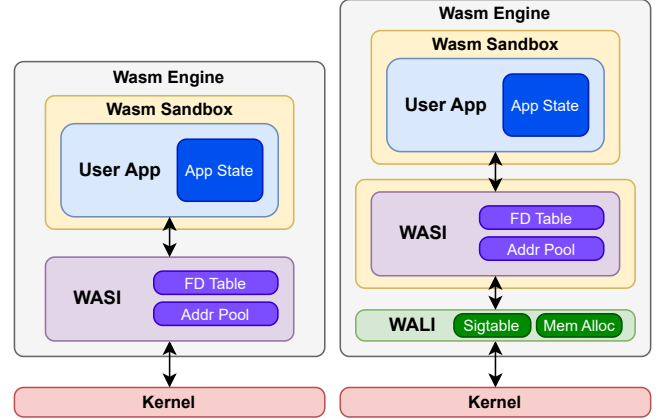

Figure 6. Minimal WALI implementation virtualizes the WASI API.

and their functionality. WALI addresses these challenges with the following techniques:

Name-bound syscalls WALI enables cross-ISA portability using *name-bound syscalls* with statically defined type signatures. This creates a clear distinction between the capabilities of the platform and the WALI implementation, allowing the latter to trap if it cannot faithfully attempt the execution of a call. The set of *virtual syscalls* in WALI are thus a union of all syscalls across supported architectures, which serve as the single, complete WALI syscall specification. Luckily, Linux syscalls show high commonality between platforms (Sec. 2), simplifying cross-ISA portability efforts.

ISA-Specific Kernel Interfaces Syscall arguments like `kstat` and file status flags, used by all `stat`-related syscalls and file control syscalls respectively, have different byte-level representations across ISAs. WALI uses a dedicated representation for these arguments, and requires the host engine to perform layout conversion to-and-from ISA-specific representations to maintain execution consistency. `ioctl` operations also may differ across ISAs; currently we only implement WALI on `x86-64`, `aarch64`, and `riscv64`, which use identical operation values. Fortunately, such disparities across ISAs are rare and require only a few lines of code to translate.

3.6 Security Model

While novel security policies are at the forefront in all Wasm APIs today, this limits the engine's ability to port many existing applications and flexibility to implement new security policies. WALI adopts a different design philosophy to security enforcement: it maintains Wasm's inherent intra-process properties (Sec. 1.1) coupled with a purely descriptive translation to the underlying kernel (i.e. Linux). This design pushes complex security policies to higher layers (Fig. 6), giving them full control in a safer and ISA portable manner. As a result, WALI is exceptionally thin (≈2000 LoC) compared to

other APIs like WASI (≈6000 LoC prior to preview2), minimizing the TCB's attack surface and allowing engine developers to support numerous security policies over a single interface. With Wasm multi-memory [14], security models can also be provided an independent privileged memory space disjoint from the application and each other.

Syscall Integrity WALI binaries possess syscall integrity, explored by works such as BASTION [56], intrinsically with name-bound invocation, intrinsic CFI, and parameter type-checking guarantees from Wasm. Syscalls in WALI do not break the Wasm intra-process sandbox guarantees since:

- *All syscalls* require only `rw-` permissions to linear memory to handle data pointers, and only `r--` access to table for function pointer callback arguments.
- *No syscall* manipulates the execution stack directly.
- All instructions in WALI binaries still only index into their private linear memories.

WALI binaries can also quickly be statically validated since the import section enumerates all syscalls the binary can potentially use up front, easing certification efforts.

Dynamic Policies `seccomp` [38] policies are commonly used to restrict applications' syscall access capabilities for software virtualization. WALI does not implement `seccomp`, but rather relies on layering to implement `seccomp-like` policies completely in user-space in the engine or as Wasm modules, making existing syscall-based security work (e.g. Draco [92]) complimentary to WALI. In the long run, security policies with a constrained set of WALI's features can serve as a simple, verifiable environment, following techniques used in security-oriented containers like Nabla [111] or gVisor [116].

Addressing Common Pitfalls Intra-process sandboxing techniques over OS abstractions are susceptible to numerous subtle security vulnerabilities [37]. We address these issues, with potential restrictions, in the context of WALI:

(1) **Filesystem Sandboxing:** Certain filesystem interfaces, notably `/proc/self/mem`, grant callers access the host process's virtual address space. WALI explicitly prevents this by interposing on all `open`-like syscalls with a explicit check for the aforementioned endpoint.

(2) **Memory Mapping:** Memory-mapping calls including `mmap`, `mremap`, and `process_vm_{read,write}v` are often exploited to generate custom executable code segments. In WALI, these attacks are *impossible* since memory is non-executable in Wasm and all mappings are sandboxed within linear memory.

(3) **Non-Local Gotos:** C/C++ features like `setjmp/longjmp` would violate traditional Wasm CFI, performing irrevocable changes to the system stack, and are thus not supported in WALI. Eventually, these features can be implemented safely transparently in toolchains as the Wasm exception-handling proposal [3] is now part of the standard.

(4) **Signal Trampoline:** The `sigreturn` syscall is often exploited as a gadget for attacks without injecting new code [33]. In WALI, however, signal handler execution is fully managed within the engine, allowing us to prohibit `sigreturn` from being directly invoked from within the WALI module user programs with a trap.

(5) **Engine Restrictions:** WALI inherently provides maximum flexibility, but some limitations may arise from the engine's internal implementation strategies. For example, engines that use signals to trigger traps (e.g. `SIGSEGV`, `SIGFPE`) might restrict the ability of WALI applications to override their respective handlers, or must implement chaining.

(6) **Processor-Specific Functionality:** Direct hardware access and the use of `ucontext/mcontext` are not supported in favor of ISA portability and security.

4 Evaluation of WALI

We evaluated WALI by compiling and executing several real-world applications, build systems, and libraries. We find that this enables Wasm binaries to more easily plug into existing ecosystems with both minimal code changes and minimal API-intrinsic overhead.

Implementation Choices We evaluate WALI with a reference implementation in the WebAssembly Micro Runtime (WAMR) [13], a popular Wasm engine written in C that supports many architectures, has extensive functionality, and has a high-performance AoT compiler in addition to an interpreter. For simplicity and completeness, we implement the *1-to-1* process model and support asynchronous signals by inserting safepoints for signal polling at loop headers for low overhead. Our WALI implementation in WAMR currently supports `x86-64`, `aarch64`, and `riscv-64` host ISAs and was deployed successfully across a cluster of 24 diverse edge devices, including 10 resource-constrained single-board computers. All evaluations on WALI below, unless otherwise specified, are collected using a fully-featured runtime on AoT compiled code.

Coverage Using our diagnostic analysis (Fig. 2), we implemented the 137 most common syscalls that cover a wide range of applications compiled against WALI to date. The WALI implementation is ≈ 2000 lines of C code, with < 100 lines of platform-specific code. We created a lightly-modified version of `musl-libc` [6] to serve as the WALI C standard library with these notable features:

- full support for threads and TLS;
- portable versions of architecture-specific structures and flags (`kstat`, file-creation flags, `ksigaction`, etc);
- static linking only, since dynamic linking is not currently supported by the Wasm ecosystem.

Codebase	Description	API portability			Missing Features
		WALI	WASIX	WASI	
bash	Shell	✓	✓	✗	signals
lua	Interpreter	✓	✓	✗	dup
virgil	Compiler	✓	✗	✗	chmod
wizard	WASM Engine	✓	✗	✗	self-host
memcached	System Daemon	✓	✗	✗	mmap
openssh	System Services	✓	✗	✗	users
sqlite	Database	✓	✗	✗	mremap
paho-mqtt	MQTT App	✓	✓	✗	sockopt
make	CLI Tool	✓	✗	✗	wait4
vim	CLI Tool	✓	✗	✗	mmap
wasm-inst	CLI Tool	✓	✗	✗	sysconf
libuvwasi	WASI Lib	✓	✗	✗	ioctl
zlib	Compression Lib	✓	✓	✓	—
libevent	System Lib	✓	✗	✗	socketpair
libncurses	System Lib	✓	✗	✗	pgroups
openssl	Security Lib	✓	✗	✗	ioctl
LTP	Test Harness	✓	✗	✗	linux

**Table 1.** Porting effort of Wasm APIs for some popular applications

We also build LLVM's `libc++` over `musl-libc` to target C++ applications and a new Rust target toolchain to support the Rust ecosystem (`wasm32-wali-linux-musl`).

4.1 Porting Effort

We collected a suite of popular Linux applications across various domains that compile using a custom WALI clang target which is similar to the existing WASI target (Table 1). Every application compiles successfully and *nearly* every one of them executes faithfully without any source code modification. The notable exception to faithful execution arises from *runtime traps* caused by failed Wasm `call_indirect` signature checks, which occur in C applications that perform function invocation with incompatible function pointer types (e.g. bash), which is actually undefined behavior. Interestingly, this outcome reflects a positive aspect of WALI 's porting process, as it can help expose type safety bugs that may be latent in low-level system software.

Surprisingly, our WALI-enabled LLVM toolchain also seamlessly integrates into complex build systems. For example, Linux allows registering interpreters for custom binary formats, enabling WALI `.wasm` files to be directly executable. This allows many build scripts to be used directly without modification[7]. The custom interpreter mechanism allowed building the entire `libuvwasi` implementation unmodified (passing all tests) and many of the currently supported syscall tests in Linux Test Project (LTP) [1] along with their test harnesses, which uses complex signalling and shared memory for job control.
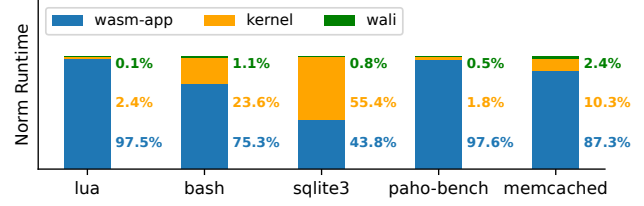
Figure 7. Runtime breakdown of WALI across system stack.

4.2 Intrinsic Costs

The performance of a WALI implementation is highly dependent on the underlying Wasm engine, which can vary drastically in performance based on how quickly it executes Wasm bytecode via interpretation or compilation [99]. While bytecode execution speed is important, it is independent of the *intrinsic cost* of using WALI that does not scale proportionally with improvements in Wasm runtimes. Our experiments with our prototype implementation on WAMR shed light on this overhead, which is mostly independent of Wasm engine. Experiments were run on a 11th Gen Intel Core i7-1185G7 machine (x86-64). However, since architecture-specific code is minimal and infrequently executed, the intrinsic cost measured here is fairly consistent across ISAs on macrobenchmarks.

Syscall Interface Most WALI syscalls require under 10 lines of code to implement – mostly performing basic address-space translation [8] – and have an absolute overhead vs native syscalls in the order of a *few hundred nanoseconds* (Table 2). To put these overheads into context, less than 1% of the execution time is typically spent in the WALI interface, which is negligible compared to the inherent overheads of the Wasm app or kernel time (Fig. 7). The `memcached` benchmark incurs a slightly higher overhead of 2.4% from WALI– we attribute this to extensive multithreading employed by the benchmark.

The `clone` syscall for spawning threads is a clear outlier, which adds about $500\mu s$ of overhead. This is however not an API-intrinsic cost of WALI, but rather that of the internal implementation of WAMR's thread manager which creates an entirely new copy of the Wasm module's execution environment for each new thread. This cost for WALI can be made significantly cheaper with various runtime optimizations – e.g, the Wasmtime [10] engine has optimized instance creation heavily through lazy loading and copy-on-write paging optimizations, resulting in overheads as low as at *5µs*. Despite this, our experience shows that most applications for our intended use-cases are not critically affected by this overhead since `clone` often occur mostly during initialization and is relatively infrequent.

---

[7]Amusingly, the *bash* build executes intermediate WALI binaries to configure certain parameters (e.g. pipe size capacity), which run transparently without modification. This is quite common in many complex builds.

[8]Calls like `rt_sigaction` and `mmap` typically need extra instructions to manage internal state for signal handling and memory allocation respectively, incurring a higher cost. These calls are the exception, not the norm.

Syscall	Overhead	LOC	State	Syscall	Overhead	LOC	State	Syscall	Overhead	LOC	State
read	167 ns	4	N	stat	112 ns	8	N	geteuid	123 ns	1	N
write	151 ns	5	N	futex	141 ns	6	N	poll	128 ns	12	N
mmap	512 ns	30	Y	rt_sigprocmask	114 ns	5	N	getrusage	151 ns	5	N
open	156 ns	4	N	getpid	168 ns	1	N	getegid	164 ns	1	N
close	187 ns	3	N	writev	387 ns	10	N	getgid	165 ns	1	N
fstat	171 ns	4	N	munmap	246 ns	12	Y	lstat	142 ns	6	N
mprotect	120 ns	4	N	fcntl	160 ns	10	N	ioctl	127 ns	4	N
pread64	671 ns	4	N	access	202 ns	8	N	clone	554873 ns	100+	Y
lseek	178 ns	3	N	recvfrom	116 ns	8	N	prlimit64	139 ns	5	N
rt_sigaction	711 ns	40	Y	getuid	151 ns	1	N	fork	345 ns	1	N

Table 2. WALI implementation statistics for 30 representative syscalls guided by heuristics from Fig 2, indicating the overhead (measured with VDSO-based `clock_gettime` of `CLOCK_MONOTONIC_RAW`), implementation size (**LOC** — Lines of Code), and whether the syscall is stateful. The high overhead of `clone` is notably a by-product of the engine rather than WALI itself.

(a) Memory – Combined   **(b)** Runtime – Lua   **(c)** Runtime – Bash   **(d)** Runtime – Sqlite
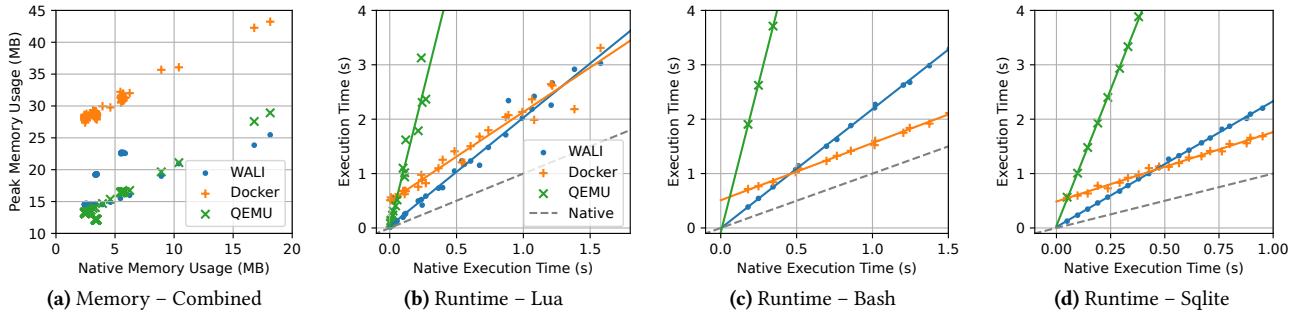
Figure 8. Peak memory (8a) and execution time (including startup time) (8b-8d) comparison for Lua, Bash, and Sqlite benchmarks showing each virtualization method's efficiency versus its native counterpart; all three benchmarks are combined into a single plot for peak memory.

App	Loop (%)	Function (%)	All (%)
bash	7.1	10.0	187.0
lua	4.1	2.8	100.3
sqlite3	11.3	5.2	164.2
paho-bench	0.5	1.1	17.8

Table 3. Cost of polling for asynchronous signal handling in WALI with different safe-point insertion schemes – **Loop**: after loop bytecode, **Func**: start of every function, **All**: after every instruction

Asynchronous Signal Polling The number of executed safepoints plays a critical role in execution overhead. Unsurprisingly, we find that polling after every instruction is prohibitively expensive–at least 10x slower than polling at loops or functions (Table 3). The latter two are comparable, typically incurring under 10% slowdown over WALI without signal polling. Both are also reasonable choices in practice – the function scheme may favor compiler optimizations better while the loop scheme may enable more reactive signal handling for large functions.

4.3 Extrinsic Costs: Virtualization Overhead

To put WALI into perspective with some existing virtualization technologies, we evaluate the performance and memory usage of our WALI implementation against two other

systems — Docker [76] for *OS interface virtualization* and QEMU [31], specifically without KVM [62], for *ISA virtualization*. We exclude KVM-based solutions (including Firecracker [24]) since KVM utilizes hardware-assisted virtualization, diverging from pure ISA virtualization that we aim to evaluate. We select three popular Linux applications in edge systems – *bash*, *lua*, and *sqlite* – to compare and contrast the optimal use cases for WALI with the above virtualization mechanisms.

WebAssembly Runtime Overhead For CPU-bound applications, the overhead of WALI versus native applications is dominated by the Wasm engine. A plethora of works have studied both execution and memory overheads of Wasm [93, 100, 106, 107], including complex optimization techniques for startup [94] and bounds-checking [97, 114], which is orthogonal to this work. To provide a baseline, an analysis in 2023 of Wasm runtime performance [42] shows a median slowdown (on performance-focused Wasm engines) of 2.32 times over native execution without SIMD [11] and Tail Call [17] language extensions.

Memory While peak memory utilization scales similarly for all virtualization solutions, base memory utilization can vary drastically (Fig. 8a). Unlike WALI, which only virtualizes the

target application, Docker containers incur a high base overhead (≈30 MB) to support intermediate layers for storage drivers and isolated software libraries. On the other hand, QEMU maintains a low overhead using a several optimizations – lazy allocation, balloon drivers, and KVM virtualization – leading to comparable results to WALI for small applications.

Execution Time Fig. 8b-8d compares the composite execution time of WALI, Docker, and QEMU. As expected of emulators, QEMU is an order of magnitude slower than Docker, which executes at near-native speed directly on the CPU. While our WALI implementation's runtime performance (slope of the line) is nearly 2*x* slower than native code and Docker on average, the startup time is only a few milliseconds as opposed to nearly half a second for containers, which requires instantiation of internal layers and `namespace` isolation. We observe a *cross-over point* for each application based on start-up time and relative overheads, before which WALI is faster than Docker. Applications with short-lived execution or those like *lua*, which execute up to 60% slower than native in Docker due to frequent memory allocation requests, are hence good candidates for using WALI as a virtualization solution.

Summary WALI strikes a middle ground in memory and execution overhead between emulation and container technologies, providing the best of both worlds: rapid startup times comparable to emulation and faster runtime performance comparable to containers. Furthermore, non-addressable execution state, ISA-portability, and CFI are guaranteed by Wasm's execution model, providing security benefits beyond default containerization techniques. While Wasm, as a relatively new execution platform, still incurs high runtime overheads (consistent with state-of-the-art analysis), we envision that increased proposal standardization [11, 17, 19, 21] and compiler/engine enhancements [100, 114] will soon close its performance gap with native execution. This will further expand WALI's reach as a feasible virtualization technique for compute-intensive workloads.

5 Kernel Interfaces Beyond Linux

Our initial design work to fit WALI into the Wasm execution model faced core challenges such as memory-mapping, process/threads, filesystems, networking etc. that are applicable to most other kernels on modern hardware. In particular, we formulate a *recipe* for designing thin kernel interfaces for beyond Linux, which entails the following:

(1) Enumerate and name-bind all OS calls that perform hardware-aided privilege escalation (e.g. `syscalls`).
(2) Sandbox (address-space translation + bounds-check) all memory addresses passed between application and OS.
(3) Encode ISA-portable struct layouts and translate to/from hardware ISA struct layouts at syscall boundaries.

(4) Map the native process model onto Wasm with sandboxed processes and/or multiple threads.
(5) Map kernel memory management primitives safely onto Wasm's linear memory model.
(6) Map asynchronous OS interactions (e.g. signal delivery/handling, RPCs) onto synchronous Wasm interactions at safepoints.

This concrete recipe allowed us to easily *auto-generate* a majority (>85%) of the WALI implementation since most calls are passthrough, only requiring steps (1)-(3). Our brief investigation shows this design methodology is applicable to syscall interfaces in most operating systems.

5.1 Applying our Recipe to Zephyr RTOS: WAZI

To validate our recipe, we perform a small prototype implementation of our recipe for a specialized RTOS, Zephyr, used extensively for IoT [65, 88], automotive [41], and industrial automation [53] domains, in WAMR. Zephyr's syscall interface (≈ 520 syscalls) is already ISA-portable (Sec. 2), and the compiler parses and creates an encoding of all syscalls at compile-time. We extract this encoding and use it for auto-generating the implementation in WAMR, much like in WALI. Our WAZI implementation is ≈ 4200 LOC.

We develop a toolchain around `picolibc`, an OS independent libc for embedded systems, and utilize WAZI to create hooks around the required I/O and filesystem calls. In addition to simple tests, we successfully compile a *Lua* binary toolchain and deploy it on a Nucleo-F767ZI ARM microcontroller board (384 kB SRAM) running Zephyr. Given WAMR is yet to fully support WASI in Zephyr, we see this as a major step towards allowing engines to target an OS like Zephyr.

5.2 Exploration of Other Mainstream Kernels

Unix-Like Kernels Most Unix-like kernels such as FreeBSD, HP-UX, and Illumos show striking similarities to Linux in their syscall set, and can follow the same design strategy as WALI. MacOS X, based on the Darwin kernel, marries a UNIX-style syscall set based on BSD with the Mach kernel syscalls, and implements POSIX compatibility via configurability in libc. While enabling a modular approach with IPC for finer-grained safety and sharing, Mach still uses WALI's core foundations albeit needing careful delineation for a kernel interface – an orthogonal program to that of WALI.

Windows The Windows kernel's syscalls evolve more rapidly than those of other systems, often being completely renumbered from one release to the next, with portability typically handled at the DLL level. While a dedicated Windows interface would likely require a stable, restricted, and portable feature subset such as the Drawbridge [84] ABI, Windows systems can presently leverage WSL2 [91] to run WALI applications unmodified.

6 Discussion and Future Outlook

Thin-kernel interfaces unlock a number of fruitful future directions by augmenting unmodified existing software with portability and sandboxing. We consider a number of future directions.

Accelerating WASI development and adoption The proliferation of WASI (especially *preview2*) is critically bottlenecked by engine implementation effort, with fully-featured support on only one engine (Wasmtime) and via a polyfill to JavaScript (`jco`). Prior to kernel interfaces, WASI's complex security model was *necessarily* part of the engine implementation. Now, with WASI decoupled from from engine development, a new standardized reference implementation could be deployed as a Wasm module that uses WALI and run on any engine. This portability greatly accelerates the evolution and adoption of WASI on new platforms.

Robustness by ecosystem modularization Typical WASI implementations themselves contain many thousands of lines of code. Vulnerabilities in *any* of this code could compromise the memory safety of Wasm and indeed, of the entire process. In contrast, WALI's *thin* syscall interface layer pushes more responsibility outside the trusted runtime system, reducing engine implementation complexity, increasing API stability, and sandboxing higher level APIs above the engine.

Portable Packaging of Linux Distributions Linux distributions offer pre-compiled packages for specific ISAs, which are compact, stable, and negate the need for complex build environments. To achieve ISA-level portability, fat binaries and multi-arch Docker images have fallen short due to unwieldy disk space overhead. In contrast, WALI executables are both compact and portable across CPU architectures; this raises the exciting prospect that Linux distributions could ultimately achieve full ISA portability with Wasm.

Full-Stack Software Verification Verification of native binaries [50] underpinned by formally-specified instruction sets [26, 89] and syscall semantics [98] have recently show great promise. Similar efforts directed towards Wasm with machine-checked proofs of modules [87], fully-verified kernels [63], compilers [68], and libraries [118] show promise to achieve the holy grail of verification: a tower of proofs to certify a program's entire software stack.

Improving Language Targetability We maintain that Wasm is an abstraction over hardware, rather than a specific security model for systems. Programming languages that target Wasm should enjoy memory safety but allow full feature-completeness, which often include low-level system calls. As more languages target Wasm, they cannot remain at the mercy of only what Web APIs or WASI allow; there must be flexibility to allow custom security layers higher in the stack to define abstractions that make the low-level interface usable, convenient, and safer.

Expansion and Interposition of Syscalls WALI already implements most "high-importance" Linux syscalls [66] including core OS features that require a custom bridge to Wasm. Our analysis of the remaining Linux API suggests that most can be added as simple pass-through calls (Sec. 5). Additionally, calls through Wasm can easily be interposed on by libraries that log, restrict, profile, or fault-inject. Unlike native syscalls which are specified by a runtime syscall number, Wasm syscalls are bound by name, allowing uniform ISA-agnostic static and dynamic policies in the future. Many tools aimed at enhancing security at the syscall layer, e.g. Nabla [111], gVisor [116], `seccomp` [38] and Draco [92], are hence complementary to this work to enable a restricted subset of secure interfaces.

7 Related Work

We organize our discussion of virtualization technologies into four broad areas: (1) Emulators, (2) Hypervisors, (3) OS interface virtualization, and (4) Language virtualization.

Emulators ISA emulators provide a mechanism for virtualizing an entire system stack including hardware, operating system, and application. Popular solutions like QEMU [31] and Bochs [64] have sparked further research into emulator performance optimizations [54] for niche use-cases [109], which is currently an obstacle to widespread adoption. These are mostly used as prototyping tools, unless KVM [62] is used when ISA emulation is unnecessary, since most binaries can run anywhere as-is, but unlike WALI, is challenging to extend to high-performance resource-constrained systems.

Hypervisors Hypervisor technology virtualizes the guest operating system kernel with bare-metal (type-1) hypervisors (e.g. vSphere [51], Xen [36]) used in cloud settings and hosted (type-2) hypervisors (e.g. Fusion [45]) are used by end users. Hyper-V [77] and KVM are common in-built type-1 hypervisors commonly leveraged in modern OSes for high performance virtualization (e.g. WSL2 [91], Firecracker [24]). In embedded domains, real-time hypervisors [57, 81, 82] are promising for cost reduction and improved resource utilization, with bolstered security from using ARM TrustZone [83]. Similar approaches that leverage hardware techniques for lightweight sandboxing [48] can also enable WebAssembly performance improvements.

OS Interface Virtualization Wine [25] offers a system compatibility layer for Windows applications to run on Linux but lacks any security advantages. Containers technologies like LXC [73] and Docker [76] are popular in cloud ecosystems for high-performance isolation that uses `namespace` and `cgroups` to control resources and isolate applications. [35] studied Docker performance in detail, reporting between 10% and 30% overhead for disk I/O and 5-10% overhead when enforcing CPU quotas. Optimizing resource isolation [105], startup time [47, 74], and heterogeneous platforms [46] has

also been a large focus of the container ecosystem for niche use-cases. Security oriented container such as Nabla containers [111] and gVisor [116] can also be mimiced via security models over WALI/WAZI. Wasm-based virtualization, however, provides CFI and RCE protection, along with ISA portability; advantages not currently available with containers.

Application Virtualization In the same vein as Wasm, numerous languages like Java, Javascript, Python, and .NET offer application-level virtualization. Browsix [85] was the first POSIX-like API for in-browser Javascript applications, emulating filesystem and sockets but pays a high performance inefficiency penalty. [95] proposed a Java OSEK interface for embedded devices and [113] a Java-based device driver virtualization, but both possess non-determinism and high memory overheads for the edge. .NET is effective in the cloud [103] but unsuitable in edge contexts [72] for similar reasons. In the Wasm ecosystem, research efforts beyond WASI(X) [49, 69, 69, 79] are directed towards designing effective edge platforms and techniques to improve security for the Wasm ecosystem [60, 67, 96], which are complementary to kernel interfaces. Native Client (and PNaCL) [44, 115] preceded Wasm for browser sandboxing, but LLVM IR is unstable, lacks full ISA portability, and contains non-deterministic behavior. With growing interest in deeply embedded [75, 104] Wasm runtimes, imminent domain-specific APIs will benefit from being virtualized over WALI.

8 Conclusion

This paper introduced the first thin kernel interfaces for Wasm which allow high-level API evolution to be decoupled from engine and bytecode evolution, improving security, robustness, and feature-completeness through layering. We show a repeatable recipe across diverse kernels, with examples WALI/WAZI, that offers a strategy for developing complete, simple, and efficient thin kernel interfaces. Wasm engines can now more easily focus on their strengths: running bytecode fast and safely exposing thin kernel interfaces, while higher-level software layers abstract over them with new security models. We believe this will unlock Wasm's portable software-defined ISA to expand beyond the Web or Cloud, supporting new low-level ecosystems while improving the development, distribution, and adoption of both WASI and new high-level APIs as Wasm code.

9 Acknowledgements

References

[1] 2012. Linux Test Project. https://github.com/linux-test-project/ltp. https://github.com/linux-test-project/ltp

[2] 2018. WasmCrypto: A WebAssembly set of cryptographic primitives. https://github.com/jedisct1/wasm-crypto. https://github.com/jedisct1/wasm-crypto (Accessed 2025-02-21).

[3] 2019. WebAssembly Exception Handling Proposal. https://github.com/webassembly/exception-handling. https://github.com/webassembly/exception-handling (Accessed 2025-02-20).

[4] 2020. The edge of the multi-cloud. https://www.fastly.com/cassets/6pk8mg3yh2ee/79dsHLTEfYIMgUwVVllaa4/5e5330572b8f317f72e16696256d8138/WhitePaper-Multi-Cloud.pdf. https://www.fastly.com/cassets/6pk8mg3yh2ee/79dsHLTEfYIMgUwVVllaa4/5e5330572b8f317f72e16696256d8138/WhitePaper-Multi-Cloud.pdf (Accessed 2021-07-06).

[5] 2020. libuvwasi. https://github.com/nodejs/uvwasi.git. https://github.com/nodejs/uvwasi.git (Access 2023-8-01).

[6] 2020. musl-libc. https://www.musl-libc.org. https://www.musl-libc.org (Accessed 2023-8-08).

[7] 2020. Unity: Getting started with WebGL development. https://https://docs.unity3d.com/Manual/webgl-gettingstarted.html. https://https://docs.unity3d.com/Manual/webgl-gettingstarted.html (Accessed 2025-02-12).

[8] 2021. Docker. https://www.docker.com/. Accessed: 2025-02-21.

[9] 2021. Wasmer: A Fast and Secure WebAssembly Runtime. https://github.com/wasmerio/wasmer. https://github.com/wasmerio/wasmer (Accessed 2025-02-18).

[10] 2021. Wasmtime: a standalone runtime for WebAssembly. https://github.com/bytecodealliance/wasmtime. https://github.com/bytecodealliance/wasmtime (Accessed 2025-02-20).

[11] 2021. WebAssembly 128-bit packed SIMD Extension. https://github.com/WebAssembly/simd/blob/main/proposals/simd/SIMD.md. https://github.com/WebAssembly/simd/blob/main/proposals/simd/SIMD.md (Accessed 2025-2-21).

[12] 2021. WebAssembly specifications. https://webassembly.github.io/spec/. https://webassembly.github.io/spec/ (Accessed 2025-02-16).

[13] 2022. WebAssembly Micro Runtime (WAMR). https://github.com/bytecodealliance/wasm-micro-runtime. https://github.com/bytecodealliance/wasm-micro-runtime (Accessed 2025-02-19).

[14] 2022. WebAssembly Multi Memory Proposal. https://github.com/WebAssembly/multi-memory. https://github.com/WebAssembly/multi-memory (Accessed 2023-7-13).

[15] 2023. CVE-2023-38408. https://www.cve.org/CVERecord?id=CVE-2023-38408. https://www.cve.org/CVERecord?id=CVE-2023-38408 (Accessed 2023-08-9).

[16] 2023. regreSSHion: CVE-2024-6387. https://www.cve.org/CVERecord?id=CVE-2024-6387. https://www.cve.org/CVERecord?id=CVE-2024-6387 (Accessed 2024-09-16).

[17] 2023. WebAssembly Tail Call Proposal. https://github.com/WebAssembly/tail-call. https://github.com/WebAssembly/tail-call (Accessed 2025-2-18).

[18] 2024. WebAssembly Custom Page Sizes Proposal. https://github.com/WebAssembly/custom-page-sizes. https://github.com/WebAssembly/custom-page-sizes (Accessed 2025-2-20).

[19] 2024. WebAssembly Garbage Collection Proposal. https://github.com/WebAssembly/gc. https://github.com/WebAssembly/gc (Accessed 2025-2-20).

[20] 2024. WebAssembly Memory-64 Proposal. https://github.com/WebAssembly/memory64. https://github.com/WebAssembly/memory64 (Accessed 2025-2-21).

[21] 2024. WebAssembly Relaxed SIMD Proposal. https://github.com/WebAssembly/relaxed-simd. https://github.com/WebAssembly/relaxed-simd (Accessed 2025-2-21).

[22] 2025. WebAssembly Shared-Everything Threads Proposal. https://github.com/WebAssembly/shared-everything-threads/tree/main. https://github.com/WebAssembly/shared-everything-threads/tree/main (Accessed 2025-2-21).

[23] Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2009. Control-Flow Integrity Principles, Implementations, and Applications. 13, 1, Article 4 (nov 2009), 40 pages. https://doi.org/10.1145/1609956.1609960

[24] Alexandru Agache, Marc Brooker, Alexandra Iordache, Anthony Liguori, Rolf Neugebauer, Phil Piwonka, and Diana-Maria Popa. 2020. Firecracker: Lightweight virtualization for serverless applications. In *17th USENIX symposium on networked systems design and implementation (NSDI 20)*. 419–434.

[25] Bob Amstadt and Michael K Johnson. 1994. Wine. *Linux Journal* 1994, 4es (1994), 3–es.

[26] Alasdair Armstrong, Thomas Bauereiss, Brian Campbell, Alastair Reid, Kathryn E. Gray, Robert M. Norton, Prashanth Mundkur, Mark Wassell, Jon French, Christopher Pulte, Shaked Flur, Ian Stark, Neel Krishnaswami, and Peter Sewell. 2019. ISA Semantics for ARMv8-a, RISC-v, and CHERI-MIPS. *Proc. ACM Program. Lang.* 3, POPL, Article 71 (jan 2019), 31 pages. https://doi.org/10.1145/3290384

[27] WASI authors. 2023. WASI: The WebAssembly System Interface. wasi.dev. https://wasi.dev/ (Accessed 2023-8-08).

[28] Wasmer authors. 2023. WASIX: The Superset of WASI. wasix.org. http://wasix.org (Accessed 2023-8-08).

[29] Awais Aziz Shah, Giuseppe Piro, Luigi Alfredo Grieco, and Gennaro Boggia. 2021. A quantitative cross-comparison of container networking technologies for virtualized service infrastructures in local computing environments. *Transactions on Emerging Telecommunications Technologies* 32, 4 (2021), e4234. https://doi.org/10.1002/ett.4234 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/ett.4234

[30] Nilanjana Basu, Claudio Montanari, and Jakob Eriksson. 2021. Frequent background polling on a shared thread, using light-weight compiler interrupts. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. 1249–1263.

[31] Fabrice Bellard. 2020. QEMU: A generic and open source machine emulator and virtualizer. http://qemu.org. http://qemu.org (Accessed 2023-8-07).

[32] Jay Bosamiya, Wen Shih Lim, and Bryan Parno. 2022. Provably-Safe Multilingual Software Sandboxing using WebAssembly. In *Proceedings of the USENIX Security Symposium*.

[33] Erik Bosman and Herbert Bos. 2014. Framing signals-a return to portable shellcode. In *2014 IEEE Symposium on Security and Privacy*. IEEE, 243–258.

[34] Emiliano Casalicchio and Stefano Iannucci. 2020. The state-of-the-art in container technologies: Application, orchestration and security. *Concurrency and Computation: Practice and Experience* 32, 17 (2020), e5668. https://doi.org/10.1002/cpe.5668 arXiv:https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.5668 e5668 cpe.5668.

[35] Emiliano Casalicchio and Vanessa Perciballi. 2017. Measuring Docker Performance: What a Mess!!!. In *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion* (L'Aquila, Italy) (*ICPE '17 Companion*). Association for Computing Machinery, New York, NY, USA, 11–16. https://doi.org/10.1145/3053600.3053605

[36] David Chisnall. 2008. *The definitive guide to the xen hypervisor*. Pearson Education.

[37] Emma Connor, Tyler McDaniel, Jared M Smith, and Max Schuchard. 2020. {PKU} pitfalls: Attacks on {PKU-based} memory isolation systems. In *29th USENIX Security Symposium (USENIX Security 20)*. 1409–1426.

[38] Jonathan Corbet. 2009. Seccomp and sandboxing. LWN (13 May 2009).

[39] Benoit Daloze, Chris Seaton, Daniele Bonetta, and Hanspeter Mössenböck. 2015. Techniques and Applications for Guest-Language Safepoints. In *Proceedings of the 10th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems* (Prague, Czech Republic) (*ICOOOLPS '15*). Association for Computing Machinery, New York, NY, USA, Article 8, 10 pages. https://doi.org/10.1145/2843915.2843921

[40] Mathias Danzeisen. 2023. Truly portable Vehicle Applications using Webassembly & WASI. (27 April 2023). https://wiki.covesa.global/display/WIK4/COVESA+All+Member+Meeting+~+April+25-27%2C+2023 COVESA All Member Meeting.

[41] Pedro Miguel Veiga de Almeida et al. 2023. Study and Implementation of Modular Software Architectures based on Hypervisors for Automotive Electronic Control Units. (2023).

[42] Frank Denis. 2023. Performance of WebAssembly runtimes in 2023. https://00f.net/2023/01/04/webassembly-benchmark-2023/

[43] Amer Diwan, Eliot Moss, and Richard Hudson. 1992. Compiler Support for Garbage Collection in a Statically Typed Language. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation* (San Francisco, California, USA) (*PLDI '92*). Association for Computing Machinery, New York, NY, USA, 273–282. https://doi.org/10.1145/143095.143140

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

[45] Micah Dowty and Jeremy Sugerman. 2009. GPU virtualization on VMware's hosted I/O architecture. *ACM SIGOPS Operating Systems Review* 43, 3 (2009), 73–82.

[46] Dong Du, Qingyuan Liu, Xueqiang Jiang, Yubin Xia, Binyu Zang, and Haibo Chen. 2022. Serverless Computing on Heterogeneous Computers. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) (*ASPLOS '22*). Association for Computing Machinery, New York, NY, USA, 797–813. https://doi.org/10.1145/3503222.3507732

[47] Dong Du, Tianyi Yu, Yubin Xia, Binyu Zang, Guanglu Yan, Chenggang Qin, Qixuan Wu, and Haibo Chen. 2020. Catalyzer: Sub-Millisecond Startup for Serverless Computing with Initialization-Less Booting. In *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems* (Lausanne, Switzerland) (*ASPLOS '20*). Association for Computing Machinery, New York, NY, USA, 467–481. https://doi.org/10.1145/3373376.3378512

[48] Bryan Ford and Russ Cox. 2008. Vx32: Lightweight user-level sandboxing on the x86. In *2008 USENIX Annual Technical Conference (USENIX ATC 08)*.

[49] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmila Cherkasova, and Gabriel Parmer. 2020. Sledge: A Serverless-First, Light-Weight Wasm Runtime for the Edge. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) (*Middleware '20*). Association for Computing Machinery, New York, NY, USA, 265–279. https://doi.org/10.1145/3423211.3425680

[50] Shilpi Goel, Warren A. Hunt, Matt Kaufmann, and Soumava Ghosh. 2014. Simulation and formal verification of x86 machine-code programs that make system calls. In *2014 Formal Methods in Computer-Aided Design (FMCAD)*. 91–98. https://doi.org/10.1109/FMCAD.2014.6987600

[51] Forbes Guthrie, Scott Lowe, and Kendrick Coleman. 2013. *VMware vSphere design*. John Wiley & Sons.

[52] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and JF Bastien. 2017. Bringing the Web up to Speed with WebAssembly. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming*

Language Design and Implementation (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 185–200. https://doi.org/10.1145/3062341.3062363

[53] Yew Ho Hee, Mohamad Khairi Ishak, Mohd Shahrimie Mohd Asaari, and Mohamad Tarmizi Abu Seman. 2021. Embedded operating system and industrial applications: a review. *Bulletin of Electrical Engineering and Informatics* 10, 3 (2021), 1687–1700.

[54] Yabin Hu, Hai Jin, Zhibin Yu, and Hongyang Zheng. 2009. An Optimization Approach for QEMU. In *2009 First International Conference on Information Science and Engineering*. 129–132. https://doi.org/10.1109/ICISE.2009.289

[55] Rishabh Iyer, Musa Unal, Marios Kogias, and George Candea. 2023. Achieving microsecond-scale tail latency efficiently with approximate optimal scheduling. In *Proceedings of the 29th Symposium on Operating Systems Principles*. 466–481.

[56] Christopher Jelesnianski, Mohannad Ismail, Yeongjin Jang, Dan Williams, and Changwoo Min. 2023. Protect the system call, protect (most of) the world with bastion. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. 528–541.

[57] Zhe Jiang, Neil C Audsley, and Pan Dong. 2018. Bluevisor: A scalable real-time hardware hypervisor for many-core embedded systems. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 75–84.

[58] Evan Johnson, Evan Laufer, Zijie Zhao, Dan Gohman, Shravan Narayan, Stefan Savage, Deian Stefan, and Fraser Brown. 2023. WaVe: a verifiably secure WebAssembly sandboxing runtime. In *2023 IEEE Symposium on Security and Privacy (SP)*. 2940–2955. https://doi.org/10.1109/SP46215.2023.10179357

[59] Marcin Juszkiewicz. 2023. Linux system calls tables for several architectures. https://marcin.juszkiewicz.com.pl/download/tables/syscalls.html. https://marcin.juszkiewicz.com.pl/download/tables/syscalls.html (Accessed 2023-08-09).

[60] Minseo Kim, Hyerean Jang, and Youngjoo Shin. 2022. Avengers, Assemble! Survey of WebAssembly Security Solutions. In *2022 IEEE 15th International Conference on Cloud Computing (CLOUD)*. 543–553. https://doi.org/10.1109/CLOUD55607.2022.00077

[61] W. M. C. J. T. Kithulwatta, K. P. N. Jayasena, Banage T. G. S. Kumara, and R. M. K. T. Rathnayaka. 2022. Integration With Docker Container Technologies for Distributed and Microservices Applications: A State-of-the-Art Review. *Int. J. Syst. Serv.-Oriented Eng.* 12, 1 (apr 2022), 1–22. https://doi.org/10.4018/IJSSOE.297136

[62] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. 2007. kvm: the Linux virtual machine monitor. In *Proceedings of the Linux symposium*, Vol. 1. Dttawa, Dntorio, Canada, 225–230.

[63] Gerwin Klein, June Andronick, Kevin Elphinstone, Gernot Heiser, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. 2010. SeL4: Formal Verification of an Operating-System Kernel. *Commun. ACM* 53, 6 (jun 2010), 107–115. https://doi.org/10.1145/1743546.1743574

[64] Kevin P Lawton. 1996. Bochs: A portable pc emulator for unix/x. *Linux Journal* 1996, 29es (1996), 7–es.

[65] Yun-kyung Lee et al. 2018. Implementation of TLS and DTLS on Zephyr OS for IoT devices. In *2018 International Conference on Information and Communication Technology Convergence (ICTC)*. IEEE, 1292–1294.

[66] Hugo Lefeuvre, Gaulthier Gain, Vlad-Andrei Bădoiu, Daniel Dinca, Vlad-Radu Schiller, Costin Raiciu, Felipe Huici, and Pierre Olivier. 2024. Loupe: Driving the Development of OS Compatibility Layers. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1* (La Jolla, CA, USA) *(ASPLOS '24)*. Association for Computing Machinery, New York, NY, USA, 249–267. https:

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

[68] Xavier Leroy. 2009. Formal Verification of a Realistic Compiler. *Commun. ACM* 52, 7 (jul 2009), 107–115. https://doi.org/10.1145/1538788.1538814

[69] Borui Li, Hongchang Fan, Yi Gao, and Wei Dong. 2021. ThingSpire OS: A WebAssembly-Based IoT Operating System for Cloud-Edge Integration. In *Proceedings of the 19th Annual International Conference on Mobile Systems, Applications, and Services* (Virtual Event, Wisconsin) *(MobiSys '21)*. Association for Computing Machinery, New York, NY, USA, 487–488. https://doi.org/10.1145/3458864.3466910

[70] Borui Li, Hongchang Fan, Yi Gao, and Wei Dong. 2022. Bringing webassembly to resource-constrained iot devices for seamless device-cloud integration. In *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*. 261–272.

[71] Renju Liu, Luis Garcia, and Mani Srivastava. 2021. Aerogel: Lightweight Access Control Framework for WebAssembly-Based Bare-Metal IoT Devices. In *2021 IEEE/ACM Symposium on Edge Computing (SEC)*. 94–105. https://doi.org/10.1145/3453142.3491282

[72] Michael H Lutz and Phillip A Laplante. 2003. C# and the. NET framework: Ready for real time? *IEEE software* 20, 1 (2003), 74–80.

[73] LXC. 2023. LXC Introduction. https://linuxcontainers.org/lxc/introduction/. https://linuxcontainers.org/lxc/introduction/ (Accessed 2023-8-08).

[74] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. 2017. My VM is Lighter (and Safer) than your Container. In *Proceedings of the 26th Symposium on Operating Systems Principles*. 218–233.

[75] J. Menetrey, M. Pasin, P. Felber, and V. Schiavoni. 2021. Twine: An Embedded Trusted Runtime for WebAssembly. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Los Alamitos, CA, USA, 205–216. https://doi.org/10.1109/ICDE51399.2021.00025

[76] Dirk Merkel. 2014. Docker: lightweight linux containers for consistent development and deployment. *Linux journal* 2014, 239 (2014), 2.

[77] Microsoft. 2021. Hyper-V Technology Overview. https://learn.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-technology-overview. https://learn.microsoft.com/en-us/windows-server/virtualization/hyper-v/hyper-v-technology-overview (Accessed 2023-8-08).

[78] Konrad Moron and Stefan Wallentowitz. 2023. Support for Just-in-Time Compilation of WebAssembly for Embedded Systems. In *2023 12th Mediterranean Conference on Embedded Computing (MECO)*. 1–4. https://doi.org/10.1109/MECO58584.2023.10155088

[79] Manuel Nieke, Lennart Almstedt, and Rüdiger Kapitza. 2021. Edgedancer: Secure Mobile WebAssembly Services on the Edge. In *Proceedings of the 4th International Workshop on Edge Systems, Analytics and Networking* (Online, United Kingdom) *(EdgeSys '21)*. Association for Computing Machinery, New York, NY, USA, 13–18. https://doi.org/10.1145/3434770.3459731

[80] Vili-Petteri Niemelä. 2021. WebAssembly, Fourth Language in the Web. (2021).

[81] Runyu Pan, Gregor Peach, Yuxin Ren, and Gabriel Parmer. 2018. Predictable virtualization on memory protection unit-based micro-controllers. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. IEEE, 62–74.

[82] Anup Patel, Mai Daftedar, Mohamed Shalan, and M Watheq El-Kharashi. 2015. Embedded hypervisor xvisor: A comparative analysis. In *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*. IEEE, 682–691.

[83] Sandro Pinto, Jorge Pereira, Tiago Gomes, Mongkol Ekpanyapong, and Adriano Tavares. 2016. Towards a TrustZone-assisted hypervisor for real-time embedded systems. *IEEE computer architecture letters* 16, 2 (2016), 158–161.

[84] Donald E Porter, Silas Boyd-Wickizer, Jon Howell, Reuben Olinsky, and Galen C Hunt. 2011. Rethinking the library OS from the top down. In *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*. 291–304.

[85] Bobby Powers, John Vilk, and Emery D. Berger. 2017. Browsix: Bridging the Gap Between Unix and the Browser. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 253–266. https://doi.org/10.1145/3037697.3037727

[86] Daniel Price and Andrew Tucker. 2004. Solaris Zones: Operating System Support for Consolidating Commercial Workloads. In *Proceedings of the 18th USENIX Conference on System Administration* (Atlanta, GA) *(LISA '04)*. USENIX Association, USA, 241–254.

[87] Xiaojia Rao, Aïna Linn Georges, Maxime Legoupil, Conrad Watt, Jean Pichon-Pharabod, Philippa Gardner, and Lars Birkedal. 2023. Iris-Wasm: Robust and Modular Verification of WebAssembly Programs. *Proc. ACM Program. Lang.* 7, PLDI, Article 151 (jun 2023), 25 pages. https://doi.org/10.1145/3591265

[88] Rafael Raymundo Belleza and Edison de Freitas Pignaton. 2018. Performance study of real-time operating systems for internet of things devices. *IET Software* 12, 3 (2018), 176–182.

[89] Alastair Reid. 2016. Trustworthy specifications of ARM® v8-A and v8-M system level architecture. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*. 161–168. https://doi.org/10.1109/FMCAD.2016.7886675

[90] Fabian Scheidl. 2020. WebAssembly: Paving the Way Towards a Unified and Distributed Intra-Vehicle Computing-and Data-Acquisition-Platform?. In *2020 AEIT International Conference of Electrical and Electronic Technologies for Automotive (AEIT AUTOMOTIVE)*. IEEE, 1–6.

[91] Prateek Singh and Prateek Singh. 2020. Exploring WSL2. *Learn Windows Subsystem for Linux: A Practical Guide for Developers and IT Professionals* (2020), 75–98.

[92] Dimitrios Skarlatos, Qingrong Chen, Jianyan Chen, Tianyin Xu, and Josep Torrellas. 2020. Draco: Architectural and Operating System Support for System Call Security. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 42–57. https://doi.org/10.1109/MICRO50266.2020.00017

[93] Benedikt Spies and Markus Mock. 2021. An Evaluation of WebAssembly in Non-Web Environments. In *2021 XLVII Latin American Computing Conference (CLEI)*. 1–10. https://doi.org/10.1109/CLEI53233.2021.9640153

[94] William Stackenäs. 2023. *An Evaluation of WebAssembly Pre-Initialization for Faster Startup Times*. Master's thesis. KTH, School of Electrical Engineering and Computer Science (EECS).

[95] Michael Stilkerich, Christian Wawersich, Andreas Gal, Wolfgang Schröder-Preikschat, and Michael Franz. 2006. OSEK/VDX API for Java. In *Proceedings of the 3rd workshop on Programming languages and operating systems: linguistic support for modern operating systems*. 4–es.

[96] Jian Sun, DingYuan Cao, Ximing Liu, ZiYi Zhao, WenWen Wang, XiaoLi Gong, and Jin Zhang. 2019. Selwasm: A code protection mechanism for webassembly. In *2019 IEEE Intl Conf on Parallel & Distributed Processing with Applications, Big Data & Cloud Computing, Sustainable Computing & Communications, Social Computing & Networking (ISPA/BDCloud/SocialCom/SustainCom)*. IEEE, 1099–1106.

[97] Raven Szewczyk, Kimberley Stonehouse, Antonio Barbalace, and Tom Spink. 2022. Leaps and bounds: Analyzing WebAssembly's

[98] Amer Tahat, Sarang Joshi, Pronnoy Goswami, and Binoy Ravindran. 2019. Scalable Translation Validation of Unverified Legacy OS Code. In *2019 Formal Methods in Computer Aided Design (FMCAD)*. 1–9. https://doi.org/10.23919/FMCAD.2019.8894252

[99] Ben L. Titzer. 2022. A Fast In-Place Interpreter for WebAssembly. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 148 (oct 2022), 27 pages. https://doi.org/10.1145/3563311

[100] Ben L Titzer. 2024. Whose baseline compiler is it anyway?. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 207–220.

[101] Chia-Che Tsai, Bhushan Jain, Nafees Ahmed Abdul, and Donald E Porter. 2016. A study of modern linux api usage and compatibility: What to support when you're supporting. In *Proceedings of the Eleventh European Conference on Computer Systems*. 1–16.

[102] Kenton Varda. [n. d.]. WebAssembly on Cloudflare Workers. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/. https://blog.cloudflare.com/webassembly-on-cloudflare-workers/ (Accessed 2025-02-21).

[103] Christian Vecchiola, Xingchen Chu, Rajkumar Buyya, et al. 2009. Aneka: a software platform for .NET-based cloud computing. *High speed and large scale scientific computing* 18, 3 (2009), 267–295.

[104] Stefan Wallentowitz, Bastian Kersting, and Dan Mihai Dumitriu. 2022. Potential of WebAssembly for Embedded Systems. In *2022 11th Mediterranean Conference on Embedded Computing (MECO)*. IEEE, 1–4.

[105] Kun Wang, Song Wu, Kun Suo, Yijie Liu, Hang Huang, Zhuo Huang, and Hai Jin. 2023. Characterizing and optimizing Kernel resource isolation for containers. *Future Generation Computer Systems* 141 (2023), 218–229. https://doi.org/10.1016/j.future.2022.11.018

[106] Weihang Wang. 2021. Empowering Web Applications with WebAssembly: Are We There Yet?. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 1301–1305. https://doi.org/10.1109/ASE51524.2021.9678831

[107] Wenwen Wang. 2022. How Far We've Come – A Characterization Study of Standalone WebAssembly Runtimes. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. 228–241. https://doi.org/10.1109/IISWC55918.2022.00028

[108] Conrad Watt. 2018. Mechanising and Verifying the WebAssembly Specification. In *Proceedings of the 7th ACM SIGPLAN International Conference on Certified Programs and Proofs* (Los Angeles, CA, USA) *(CPP 2018)*. Association for Computing Machinery, New York, NY, USA, 53–65. https://doi.org/10.1145/3167082

[109] Ming-ting Wei, Yu-Shiang Lin, and Che-Rung Lee. 2019. Performance Optimization for InfiniBand Virtualization on QEMU/KVM. In *2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. 19–26. https://doi.org/10.1109/CloudCom.2019.00016

[110] Elliott Wen and Gerald Weber. 2020. Wasmachine: Bring IoT up to Speed with A WebAssembly OS. In *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. 1–4. https://doi.org/10.1109/PerComWorkshops48775.2020.9156135

[111] Dan Williams and Ricardo Koller. 2016. Unikernel Monitors: Extending Minimalism Outside of the Box. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*. USENIX Association, Denver, CO. https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/williams

[112] Chris Woods and Ajay Chhokra. 2022. An end-to-end toolchain for evaluating WebAssembly runtimes for CPS-IoT Use cases. (25 October 2022). https://www.cs.cmu.edu/~wasm/wasm-research-day-2022.html WebAssembly Research Day 2022.

performance with a focus on bounds checking. In *2022 IEEE International Symposium on Workload Characterization (IISWC)*. 256–268. https://doi.org/10.1109/IISWC55918.2022.00030

[113] Hiroshi Yamauchi and Mario Wolczko. 2006. Writing Solaris device drivers in Java. In *Proceedings of the 3rd workshop on Programming languages and operating systems: linguistic support for modern operating systems*. 3–es.

[114] Zachary Yedidia. 2024. Lightweight fault isolation: Practical, efficient, and secure software sandboxing. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. 649–665. https://doi.org/10.1145/3620665.3640408

[115] Bennet Yee, David Sehr, Gregory Dardyk, J Bradley Chen, Robert Muth, Tavis Ormandy, Shiki Okasaka, Neha Narula, and Nicholas Fullagar. 2010. Native client: A sandbox for portable, untrusted x86 native code. *Commun. ACM* 53, 1 (2010), 91–99.

[116] Ethan G Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. 2019. The true cost of containing: A {gVisor} case study. In *11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 19)*.

[117] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. 2022. {XRP}:{In-Kernel} Storage Functions with {eBPF}. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. 375–393.

[118] Jean-Karim Zinzindohoué, Karthikeyan Bhargavan, Jonathan Protzenko, and Benjamin Beurdouche. 2017. HACL*: A Verified Modern Cryptographic Library. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security* (Dallas, Texas, USA) *(CCS '17)*. Association for Computing Machinery, New York, NY, USA, 1789–1806. https://doi.org/10.1145/3133956.3134043

A Artifact Appendix

A.1 Abstract

The artifact accompanying this paper provides public repositories for an Ubuntu 22.04 Virtual Machine and relevant code for a full WALI ecosystem. This serves both as a means of reproducing claims made in the paper as well as a playground to further explore the capabilities of WALI. This artifact excludes WAZI in the image, due to strict hardware requirements for reproducibility – however, code artifact for WAZI is made available on GitHub.

A.2 Description & Requirements

A.2.1 How to access

The experimental setup for the artifacts is contained within a Ubuntu 22.04 VM, which is made available as a public repository on *Zenodo* (see Table. 4 for URLs). The repositories for WALI code and the WALI experiments listed in Table. 4 are also publicly available on Github and **already cloned/setup** on the VM for ease of reuse. The VM disk image is shipped in QCOW2 format its XML definition. The login information is below:

- Username: evaluator (has sudo access)
- Password: webassembly

A.2.2 Hardware dependencies

The VM should run on effectively any hardware that can support virtualization. We recommend a system with hardware-assisted virtualization for performance reasons, with at least 8-16 cores of CPU and 8-16 GB of RAM for the VM. The image itself consumes 26 GB of physical disk.

A.2.3 Software dependencies

We recommend a Linux distribution with QEMU/KVM support to run the VM, managed with virsh tools. VirtualBox may alternatively also be used, but requires converting the QCOW2 image to VDI (see https://gist.github.com/mamonu/671038b09f5ae9e034e8). The numerous software dependencies for the project have all been already added in the VM.

A.2.4 Benchmarks

The benchmarks in Table. 1 are all available in the WALI codebase, mostly as submodules, under applications. Note, the binary mqtt-app is sometimes aliased as paho-bench in the paper.

A.3 Set-up

The software environment should be mostly ready to go by default on the VM + XML definition on Zenodo. The XML file defines 8 CPUs and 8 GB of RAM – modify it as needed based on physical system constraints. You can extract and create the VM using these commands in Linux:

1. *tar –sparse -xvf vm-ubuntu22.04-wali-eurosys25.tar.gz*
2. *sudo mv u22.04-eurosys.qcow2 /var/lib/libvirt/images*
3. *virsh define u22.04-eurosys.xml*

A.3.1 VM Operation

The following are helpful commands to operate the VM with virsh:

- **Start VM**: virsh start u22.04-eurosys
- **Shutoff VM**: virsh destroy u22.04-eurosys
- **Connecting to VM**: This can be done in 3 ways:
    - Console: virsh console u22.04-eurosys
    - GUI: virt-viewer u22.04-eurosys
    - SSH to IP address of VM, obtained from console

A.3.2 VM Directories

The VM should contain three directories (including repos from Table. 4):

- WALI: The cloned repo of WALI source code
- wali-eurosys25-data: The cloned experiments repo of WALI, with packaged data used in evaluation
- artifacts: Compressed directory containing a variety of builds, described in Table. 5 for quick setup and reducing large build size/times.

Many files have been compressed to minimize VM image size, so some setup steps below are required to perform.

WALI The README.md in the WALI code repo has a setup and information guide if you intend to rebuild everything or need details. We have however provided everything built — the WALI runtime iwasm, the sysroot in wali-musl, and llvm-project/build.

The llvm-build.tar.gz is prepackaged under *artifacts* and can be extracted with *tar -xvf artifacts/llvm-build.tar.gz -C WALI/llvm-project/*. Note in case you decide to manually build – be prepared for LLVM to take up 130GB of disk and requires 8+ CPUS and 16+ GB of RAM.

We have also already registered Wasm as a miscellaneous binary format as specified in the README.md — you can just run .wasm files like ELF files (e.g. ./bash.wasm —norc)

wali-eurosys25-data The README.md in the WALI experiment repo has a detailed setup guide as well. The only step required for setup is to move the needed virt.tar.gz binaries under artifacts with *tar -xvf artifacts/virt.tar.gz -C wali-eurosys25-data/* for experiments.

A.4 Evaluation workflow

Below we highlight list the claims and required experiments needed to verify them.

A.4.1 Major Claims

- *(C1): WALI allows porting most complex Linux applications unmodified over Wasm, unlike WASI/X. This is shown by E1 described in Sec. 4.1 and Table. 1*
- *(C2): WALI allows layering complex security policies like* WASI *over it. This is shown by E2 described in Sec. 4.1 and Table. 1.*
- *(C3): WALI strikes a middle-ground between container virtualization (Docker) and ISA virtualization (QEMU), with a reasonably fast startup overhead like QEMU, and a reasonably fast execution overhead like Docker. This is shown by E3 described in Sec. 4.3 and Fig. 8.*

Artifact	URL	DOI/Hash
Zenodo Virtual Machine Distribution	https://doi.org/10.5281/zenodo.14790613	10.5281/zenodo.14790613
WALI Code Repo	https://github.com/arjunr2/WALI	1e22d2e
WALI Experiments Repo	https://github.com/arjunr2/wali-eurosys25-data	ac83120

Table 4. List of relevant URLs for artifacts along with their commit hashes

Tar File	Description
applications-artifact	A wide suite of applications that have been ported over WALI, including those in Table. 1
llvm-build	A compressed build of LLVM to reduce disk size and save on time to build LLVM
virt	Miscellaneous binaries for WALI/QEMU (iwasm, qemu, wamrc) for experiments
sysroot	(Optional) A build of wali-musl libc

Table 5. List of packaged software under `artifacts` directory in the VM

A.4.2 Experiments

Experiment (E1) *[Portability] [20 human-minutes + 5 compute minutes]:* All ported WALI Wasm apps are in *applications-artifact.tar.gz*, which can be run directly. No preparation is necessary, and the ecosystem is already setup.

[Execution]: You can run all WALI binaries like ELF binaries! Command line arguments are passed akin to normal binaries, and additionally a WALI_VERBOSE environment variable can be configured to show dynamically executed syscalls (e.g. WALI_VERBOSE=5 ./bash.wasm −norc).

[Results]: As stated in Sec. 4.1 and Table. 1, we can observe the dynamic syscalls executed that are missing from WASI/X in verbose mode – the latter's spec doesn't support these so they wouldn't compile in the first place.

Experiment (E2) *[WASI Layering] [5 human-minutes + 5 compute-minutes]:* We compile a popular implementation of WASI, libuvwasi to run over WALI completely, passing all unit tests.

[Execution]: Run the following commands:

1. cd WALI/applications
2. ./run_libuvwasi_tests.sh

[Results]: The ctest harness should execute 22 tests, passing all of them as stated in Sec. 4.1, hence realizing Fig. 1.

Experiment (E3) *[Comparison to Docker/QEMU] [15 human-minutes + 30 compute-minutes]:* Compare WALI against Docker for container virtualization and QEMU for ISA virtualization for memory and runtime overheads.

[Execution]: Follow the instructions under *Rerun Benchmarks* in the README for WALI Experiments Repo for the

relevant comparison. After executing, run ./gen_plots.sh to generate all the figures

[Results]: Since these tests are very sensitive to hardware platforms, the exact ratios may differ substantially between platforms. However, it should be consistent that WALI strikes the middle ground, with faster startup time and memory usage than Docker and faster execution time than QEMU (Sec. 4.3) – see figures memory.pdf and runtime_*.pdf, which are similar to Fig. 8)

Filename	Description
memory.pdf	Memory overhead plot (Fig. 8a)
runtime_*.pdf	Runtime overhead plots (Fig. 8b, 8c, 8d)
syscall_profile.pdf	Syscall Profile across applications (Fig. 2)
wali_macrobench.pdf	Macrobenchmark overhead split between WALI, kernel and application space (Fig. 7)
syscall_archs.pdf	Common syscalls between architectures (Fig. 3)
sigpoll.txt	Sigpoll overhead results (Table. 3)
syscall_overheads.txt	Intrinsic WALI overhead (Table. 2)

Table 6. List of generated plots from `data.tar.gz`. Generate all of these by running ./gen_plots.sh

A.5 Notes on Reusability

Generating all of the figures based on collected data (i.e those in Table. 6) is easy, and detailed in the README in WALI Experiments Repo. Setting up the entire WALI toolchain locally is also straight-forward, following the README in WALI Code Repo. We hope this encourages people to experiment and further research into Wasm over WALI.

A.6 General Notes

WALI is fully open-source and actively under development and improvement at the date of writing.