Wasmachine: Bring IoT up to Speed with A WebAssembly OS

Elliott Wen
The University of Auckland
jwen929@aucklanduni.ac.nz

Gerald Weber
The University of Auckland
g.weber@aucklanduni.ac.nz

***Abstract*—WebAssembly is a new-generation low-level bytecode format and gaining wide adoption in browser-centric applications. Nevertheless, WebAssembly is originally designed as a general approach for running binaries on any runtime environments more than the web. This paper presents Wasmachine, an OS aiming to efficiently and securely execute WebAssembly applications in IoT and Fog devices with constrained resources. Wasmachine achieves more efficient execution than conventional OSs by compiling WebAssembly ahead of time to native binary and executing it in kernel mode for zero-cost system calls. Wasmachine maintains high security by not only exploiting many sandboxing features of WebAssembly but also implementing the OS kernel in Rust to ensure memory safety. We benchmark commonly-used IoT and fog applications and the results show that Wasmachine is up to 11% faster than Linux.**

I. Introduction

WebAssembly is a young binary instruction format first released in 2017 by mainstream browser vendors [1]. Its primary design goal was to provide an approach to run portable executables with near native performance in web browser environments. Developers, who wish to leverage WebAssembly, can write their applications in various kinds of high level languages (e.g., C++ or Javascript) and use corresponding compiler toolchains to generate WebAssembly binary. Recently, we have witnessed the emergence of many web applications built on WebAssembly such as data visualization [2], video processing [3], and games [4].

More recently, researchers start taking WebAssembly beyond the web. Two research works [5], [6] have explored WebAssembly's potential, specifically security and portability, on IoT and edge computing. WebAssembly possesses many security features such as sandboxed linear memory and structured control flow. They can mitigate many common security vulnerabilities and errors stemming from direct memory access. WebAssembly is also agnostic to source languages and target platforms. Therefore, a WebAssembly binary can be compiled once and run freely across different hardware architectures without reconfiguration. Despite these promising results, a recent study [7] reveals an open challenge of using WebAssembly in IoT and fog computing: an application compiled to WebAssembly runs slower by an average of 45% compared with its native opponent (i.e., same application compiled to native machine codes). Though the study does no provide any concrete solutions to the performance parity, its analysis hints that the culprit may lie in existing WebAssembly runtime.

A conventional WebAssembly runtime, as shown in Fig I (a), is a program that translates WebAssembly binary instructions to native CPU machine codes before execution. The translation is most achieved in a just-in-time (JIT) fashion; when a WebAssembly application starts, it will be first interpreted, and after a while, methods frequently executed will be compiled to native codes to improve execution efficiency. JIT enables fast start up time but less efficient codes due to limited time that can be spent on code optimization. Using JIT is reasonable in the context of web browsing, where startup time may significantly affect user experience. However, it is suboptimal for IoT or fog computing, where code efficiency is preferred.

A runtime also assists a WebAssembly program with system call operations (e.g., networking or file access). Specifically, WebAssembly does not have built-in privileged instructions like system calls and hardware input/output. The only workaround for WebAssembly to execute system calls is to invoke WebAssembly system interfaces (WASI), a set of system call wrapper functions explicitly exposed to WebAssembly memory by the runtime. This invocation process has to go through two boundaries as shown in Fig I (a); the first one is from applications to runtime (normalizing function parameters to match system call prototypes), and the second one is from runtime to kernel (context switching from Ring 3 user mode to Ring 0 kernel mode). Either of these boundaries can incur a significant performance overhead.

To address these issues, we present Wasmachine, a secure runtime/OS aiming to run WebAssembly applications faster than native on a bare-metal machine. Figure I (b) demonstrates the basic workflow of Wasmachine. Given a number of WebAssembly applications, Wasmachine first compiles them into native binary objects using an Ahead-of-Time (AOT) compiler. AOT compilation can spend as much time applying complex optimization methods and tends to produce more efficient binary. The resulting objects can then be statically linked to Wasmachine runtime. Unlike conventional runtime programs which have to run on top of an OS, Wasmachine runtime itself is an OS kernel for bare-metal machines. Wasmachine kernel exports system calls in WASI prototypes and runs each WebAssembly application as a sandboxed kernel thread (i.e., in Ring 0). This design enables WebAssembly applications to invoke system calls as normal functions without the cost of parameter normalization and context switching. We implement the kernel in Rust, a strong-type system-level

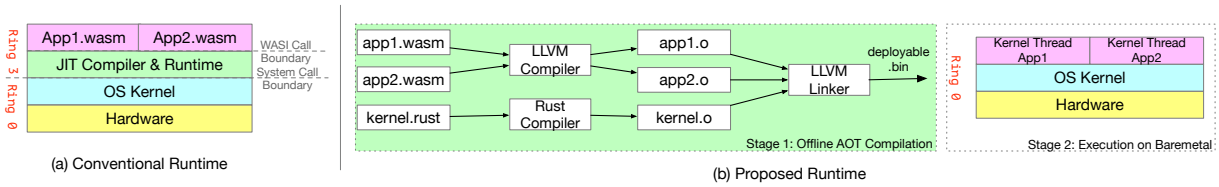

Fig. 1. System Architecture of Existing Runtime and Proposed Runtime.

programming language that delivers memory safety with zero-cost abstractions. It has a low memory footprint (102KB for x64) and can be easily deployed to embedded devices with limited resources.

We consolidated the above techniques and implemented a prototype of Wasmachine on commonly-used hardware architectures such as x64 and aarch64. Based on the prototype, we conducted a performance benchmark on several essential components in IoT and fog computing. The results show that WebAssembly applications in Wasmachine is up to 11% faster than their native opponents in Linux.

The remainder of this paper is structured as follows: Section. II explains the design of our AOT compiler for WebAssembly. Section. III elaborates the kernel implementation of Wasmachine. Section. IV offers a preliminary performance evaluation of Wasmachine. Section. V shows the related work and is followed by section. VI concluding this paper with future work.

II. AOT Compilation for WebAssembly

Wasmachine implements an AOT compiler to compile WebAssembly instructions into native CPU instructions before execution. AOT compilation brings about multiple advantages. First, AOT compilation can perform complex code optimizations, which in most cases of JIT compilation will be considered much too costly. Secondly, it can also save disk space because we do not have to ship a bulky compiler in the runtime. Lastly, AOT compilation can also improve a device's battery life by doing compute-intensive compilation process offline. These features can be quite useful for resource-constrained IoT or fog devices.

Our AOT compiler is constructed with the LLVM compiler infrastructure. We first utilize Flex and Bison to generate a lexer and a parser for WebAssembly. These tools allows us to iterate each WebAssembly instruction and convert it into LLVM intermediate representation (IR). The conversion is generally not difficult since WebAssembly and LLVM IR are both relatively low-level instructions and many of them are similar semantically. The conversion result may be suboptimal and can be significantly improved with the help of LLVM optimization passes (e.g., constant elimination and loop unrolling). Afterwards, the optimized IRs can be sent to LLVM backends to generate native objects for different CPU architectures. These native objects will be statically linked to Wasmachine kernel and function as a kernel thread.

Running untrusted applications in kernel space is generally a risky idea. Nevertheless, our AOT compiler is designed to generate trusted and sandboxed binary objects by exploiting many security features of WebAssembly. The first security feature we can harness effortlessly is that WebAssembly has no built-in privileged instructions such as hardware input/output or syscall. It ensures that WebAssembly alone (and its translated native binary) has no capability to express and execute any sensitive instructions. The only workaround to execute privileged instructions is through external APIs (WASI) explicitly exposed to WebAssembly memory space. These external APIs, however, can be easily scrutinized by our compiler in the linkage time. For instance, our compiler can whitelist socket functions to allow Internet access, while blacklisting the file functions to prevent filesystem access.

Our compiler also delivers application memory isolation by exploiting the linear memory model of WebAssembly. In details, every WebAssembly application has one specially-designated default linear memory *mem*. It is a contiguous byte-addressable range of memory spanning from offset 0 and extending up to a varying memory size. The memory content can be read and written by memory operation instructions 'load' and 'store' with an offset parameter. A simplified example is 'i32.load 10', which loads a 32-bit integer sitting in *mem[10:13]*. To translate these memory operations, our compiler first inserts a program initialization routine, which allocates a contiguous memory chunk to serve as the linear memory and records its address in a global variable $base$. Afterwards, the compiler transforms the parameter of each memory operation by adding the value of $base$. For instance, the WebAssembly instruction 'i32.load 10' will be transformed into LLVM IR '%address_temp = 10 + @base; %result = load i32, i32* %address_temp'. We can ensure that all memory operations are constrained in the allocated linear memory by injecting bounds-checking instructions (compare and branch) on '%address_temp'. If an out-of-bound access is detected, we would terminate the application immediately to ensure system security. However, this software bound-checking inevitably leads to performance penalty. Wasmachine introduces a hardware checking to avoid the penalty in the next section.

Another safety guarantee our compiler can deliver is control-flow integrity. It is a safety mechanism that prevent attackers from arbitrarily controlling program behavior (i.e., making unintended control-flow transitions). Generally, there are three types of external control-flow transitions that need to be protected including 1) Direct jumps or function calls,

2) Indirect function calls, and 3) Returns. Note that (1) has been partially protected by WebAssembly's structured control flow, which represents code as an ordered sequence of basic blocks and scoped control flow constructs (e.g., if-else or loop). Notably, goto (jump) statement is deliberately excluded and branching instructions must point to valid destinations within the enclosing constructs. Our compiler further protects (1) by setting the application's code segment to be immutable to prevent code injection. To protect 2), our compiler injects instructions to verify signature of indirect function calls at each call site. Regarding 3), our compiler employs a shadow stack technology, which stores stack variables in a separate user-addressable stack in linear memory at compile time. This prevents attackers to overwrite return address by stack smashing (e.g., stack overflow).

III. Kernel Implementation

Wasmachine implements a lightweight kernel to better fit the needs of running WebAssembly applications efficiently in resource-constrained devices. This section briefly describes the implementation of our kernel, focusing on areas in which the integration of WebAssembly affected the design.

Kernel Architecture and Programming Language. Wasmachine features a Unix-like monolithic kernel architecture and currently supports 64-bit x86 and armv8 hardware. The kernel is written in assembly and Rust. Assembly is required to initialize hardware (e.g., timer and interrupt controller) during boot time and to save/restore registers during a context switch. The reasons we use Rust rather than C are multifold. First, C requires extra care to manage memory safely, even then bugs (e.g., buffer overflow, use-after-free and data racing) are still common. Rust can easily prevent these bugs by enforcing a set of type-safety and memory-safety rules in compile time. Rust also provides many convenient abstractions such as unicode strings and memory pools, which we can reuse to reduce the implementation workload.

Process Management. Wasmachine simplifies the process management model in two asepcts. First, Wasmachine does not make use of 'protection ring'. Unlike convention OSs that run user applications and kernel codes in Ring 3 and Ring 0 separately, Wasmachine runs everything in Ring 0. Wasmachine disables the hardware protection because our compiler has already enforced the sandbox protection. This design also brings about a significant advantage: it removes the context switch overhead of each system call, which is now just a normal function call. Secondly, Wasmachine treats each WebAssembly process as a thread. This simplification is made based on the fact that WebAssembly currently does not support multithreading. We are planing to alter this process model once the multithreading feature becomes available in the next version of WebAssembly.

Memory Management. Since the kernel and WebAssembly applications are all running in Ring 0, they share a same memory address space, in other words, using a same paging table. This improves memory cache performance since it avoids page cache invalidation due to page table switching. Nevertheless, the process memory isolation can still be implicitly enforced by bounds-checking of memory operations. Wasmachine uses a hardware bounds-checking. It exploits a fact that WebAssembly memory instructions take a 32-bit unsigned integer as an offset of the linear memory (i.e., 4GB addressable space). Therefore, by arranging memory regions of processes 4GB away from each other, we can ensure that an application would not be able to access the memory from other applications. A possible implementation is that we first divide entire virtual memory space into segments of 4GB and number them. Afterwards, we can assign memory segments with an odd numbers for each application. Unlike software bounds-checking, this approach inserts no extra instructions and incurs no performance overhead.

System Calls. Wasmachine implements a subset of WASI functions as system calls. They provide filesystem access via a Virtual File System (VFS) layer that supports popular low-level filesystem formats (e.g., in-memory, EXT2, FAT32). Wasmachine also enables networking via an off-the-shelf TCP/IP stack 'Smoltcp' and implements commonly used Ethernet adapter drivers. Note that Wasmachine is SMP-aware and can run processes in parallel on multi-core hardware. To enable concurrent access to the system calls, Wasmachine implements commonly-used locking primitives like spinlocks, mutexes and semaphores to guard and synchronize internal data structures.

IV. Preliminary Evaluation

This section demonstrates preliminary performance evaluation of Wasmachine. The experiments reported below were run on a KVM machine with a quad-core 3 GHz GPU, 1 GB RAM, and a dedicated gigabit Ethernet connection. The benchmarks use an in-memory file system in order to stress the CPU efficiency of the kernel. A HTTP server and an in-memory database are tested, since they are essential components in IoT and fog computing. Meanwhile, we also evaluate a real-life use case, which is a LaTeX online compilation service.

HTTP Server. We use Nginx as a HTTP server. The server is configured with one worker. A client runs a stress testing tool called ApacheBench to execute 5000 HTTP GET requests in total and 500 requests at a time. Each request will obtain a dummy file of 1 KB.

In-memory database. Redis is an in-memory key/value database. The benchmark runs one single-threaded Redis server. A client uses a program called redis-benchmark to generate 1000000 GET requests with random keys.

LaTeX. We also evaluate a real-life use case. We deploy an online LaTeX compilation service inside our organization such that everyone can upload and compile their LaTeX documents using browsers. We implemented the service using a modified LaTeX engine which constantly fetches a compilation job from a task queue. We measure the compilation time for a 10-page IEEE LaTeX documents.

The experiments are carried out under three settings, including WebAssembly in Wasmachine, WebAssembly in conventional runtime (WasmTime), and native binary in Linux. We

TABLE I
PERFORMANCE COMPARISON IN DIFFERENT SETTINGS

	Conventional	Native	Wasmachine
Nginx	76% (31.72s)	100% (24.11s)	111% (21.73s)
Redis	79% (75.59s)	100% (59.72s)	109% (54.78s)
LaTeX	91% (1.54s)	100% (1.41s)	101% (1.39s)

present the results in Table I to allow for better comparison. It can be seen that Linux can be up to 24% faster than the conventional runtime. This result roughly matches the observations from the previous study [7]. We can also notice that Wasmachine boosts performance for all tested applications. Nginx receives the most performance gain of 11%, while there is only 1% performance improvement for LaTeX. We suspect that Nginx, as a networking application, is system-call intensive and thus benefits more from kernel mode execution than the CPU-intensive LaTeX. We are planning to conduct a detailed evaluation by benchmarking more applications and looking into how Rust code affects the overall performance.

V. RELATED WORK

WebAssembly is a low-level bytecode format, which provides an abstraction over modern hardware. It is independent to programming languages, hardware architectures, and operating systems. Today, a growing number of compilers start supporting WebAssembly as a compilation target. They can compile commonly-used programming languages such as C/C++ [8] and Javascript [9] into WebAssembly binaries, which can be directly executed in mainstream browsers. Many useful WebAssembly-based browser applications are emerging such as interactive 3D visualization [2], audio and video software [3], and games [4].

More recently, researchers are exploring WebAssembly's potential beyond the Web, especially on IoT and Fog Computing. Jacobsson et al., [5] exploits WebAssembly's portability to run the same WebAssembly binary across embedded devices with different hardware architectures. Commerical-grade Cloudflare [10] and a recent research work [6] exploit secure features of WebAssembly to enable sandbox execution of user-provided WebAssembly modules. Recently, a performance evaluation of WebAssembly [7] shows that an application compiled to WebAssembly runs slower by an average of 45% compared with its native opponent. A potential culprit is the WebAssembly runtime. Existing runtime environments translate WebAssembly instructions to native binary using JIT compilation. It tends to generate sub-optimal binary due to the lack of sufficient code optimizations. Moreover, these environments are acting as another sandboxed 'OS' on top of the real OS. This design incurs a significant overhead in performance.

VI. CONCLUSION AND FUTURE WORK

This paper presents Wasmachine, a secure OS that can efficiently execute WebAssembly applications in IoT and edge devices with constrained resources. Wasmachine is efficient thanks to its ahead-of-time compilation and kernel mode execution. Wasmachine maintains high security owing to sandbox features of WebAssembly and memory-safety in Rust.

Nevertheless, Wasmachine is still a research prototype and missing some useful features for production. One important pending feature is resource provisioning, which can be used to limit a process's execution time and maximum memory usage in multi-task scenarios. Currently, Wasmachine only applies rudimentary Robin Round process scheduling such that every process has a fair share of CPU time. Wasmachine also assumes that all processes can be fitted into main memory and is yet to implement the page swapping. Another missing feature is hardware-specific accelerations. Our AOT compiler now instructs LLVM backends to disable hardware-specific acceleration features such as AVX2 or SIMD in order to maximize binary portability. We are planing to enable acceleration extensions in next release by allowing users to specific feature profiles of target CPUs. Wasmachine is also lacking a cluster management functionality. Nowadays, near all IoT and fog computing systems work in a distributed manner. It is essential to provide a cluster management framework such that users can easily scale and manage their applications. We are planing to open-source our AOT compiler and kernel such that we can receive more contributions from enthusiast developers.

REFERENCES

[1] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and J. Bastien, "Bringing the web up to speed with webassembly," in *ACM SIGPLAN Notices*, vol. 52, no. 6. ACM, 2017, pp. 185–200.

[2] R. Finney and D. Meerzaman, "Chromatic: Webassembly-based cancer genome viewer," *Cancer informatics*, vol. 17, p. 1176935118771972, 2018.

[3] S. Letz, Y. Orlarey, and D. Fober, "Compiling faust audio dsp code to webassembly," 2017.

[4] A. Zakai, "Fast physics on the web using c++, javascript, and emscripten," *Computing in Science & Engineering*, vol. 20, no. 1, pp. 11–19, 2018.

[5] M. Jacobsson and J. Wåhslén, "Virtual machine execution for wearables based on webassembly," in *13th EAI International Conference on Body Area Networks (BodyNets 2018)*, 2018.

[6] A. Hall and U. Ramachandran, "An execution model for serverless functions at the edge," in *Proceedings of the International Conference on Internet of Things Design and Implementation*. ACM, 2019, pp. 225–236.

[7] A. Jangda, B. Powers, E. D. Berger, and A. Guha, "Not so fast: Analyzing the performance of webassembly vs. native code," in *2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19)*, 2019, pp. 107–120.

[8] A. Zakai, "Emscripten: an llvm-to-javascript compiler," in *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*. ACM, 2011, pp. 301–312.

[9] M. Reiser and L. Bläser, "Accelerate javascript applications by cross-compiling to webassembly," in *Proceedings of the 9th ACM SIGPLAN International Workshop on Virtual Machines and Intermediate Languages*. ACM, 2017, pp. 10–17.

[10] K. T. Varda, Z. A. Bloom, M. P. Majkowski, I. Stepanyan, K. Kloepper, D. O. Knecht, J. Graham-Cumming, and D. Grant, "Cloud computing platform that executes third-party code in a distributed cloud computing network," Jun. 25 2019, uS Patent 10,331,462.