# POSTER: Leveraging eBPF to enhance sandboxing of WebAssembly runtimes

Marco Abbadini
marco.abbadini@unibg.it
Università degli Studi di Bergamo
Bergamo, Italy

Michele Beretta
michele.beretta@unibg.it
Università degli Studi di Bergamo
Bergamo, Italy

Dario Facchinetti
dario.facchinetti@unibg.it
Università degli Studi di Bergamo
Bergamo, Italy

Gianluca Oldani
gianluca.oldani@unibg.it
Università degli Studi di Bergamo
Bergamo, Italy

Matthew Rossi
matthew.rossi@unibg.it
Università degli Studi di Bergamo
Bergamo, Italy

Stefano Paraboschi
stefano.paraboschi@unibg.it
Università degli Studi di Bergamo
Bergamo, Italy

## ABSTRACT

WebAssembly is a binary instruction format designed as a portable compilation target enabling the deployment of untrusted code in a safe and efficient manner. While it was originally designed to be run inside web browsers, modern runtimes like Wasmtime and WasmEdge can execute WebAssembly directly on various systems. In order to access system resources with a universal hostcall interface, a standardization effort named WebAssembly System Interface (WASI) is currently undergoing. With specific regard to the file system, runtimes must prevent hostcalls to access arbitrary locations, thus they introduce security checks to only permit access to a predefined list of directories. This approach not only suffers from poor granularity, it is also error-prone and has led to several security issues. In this work we replace the security checks in hostcall wrappers with eBPF programs, enabling the introduction of fine-grained per-module policies. Preliminary experiments confirm that our approach introduces limited overhead to existing runtimes.

## CCS CONCEPTS

• **Security and privacy → Software and application security**; **Access control**; *File system security*.

## KEYWORDS

Sandboxing, Access control, WebAssembly runtime, eBPF

**ACM Reference Format:**
Marco Abbadini, Michele Beretta, Dario Facchinetti, Gianluca Oldani, Matthew Rossi, and Stefano Paraboschi. 2023. POSTER: Leveraging eBPF to enhance sandboxing of WebAssembly runtimes. In *ACM ASIA Conference on Computer and Communications Security (ASIA CCS '23), July 10–14, 2023, Melbourne, VIC, Australia.* ACM, New York, NY, USA, 3 pages. https://doi.org/10.1145/3579856.3592831

*ASIA CCS '23, July 10–14, 2023, Melbourne, VIC, Australia*
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0098-9/23/07.
https://doi.org/10.1145/3579856.3592831

## 1 INTRODUCTION

WebAssembly (Wasm) [15] is a popular binary instruction format that enables the execution of untrusted code in a safe, isolated environment. Moreover, it is a portable compilation target for different languages, and can be executed efficiently on a wide range of platforms without the need of dedicated hardware. Wasm was originally meant to be run inside web browsers, but given the considerable advantages it brings, many runtimes that allow execution in standalone mode have been developed recently. Popular examples are Wasmtime, WasmEdge, Wasmer, and WAMR.

To answer the developers' need to access resources of the host system from within the runtime, a standardization effort called WebAssembly System Interface (WASI) [28] is undergoing. Its goal is to provide a stable and multi-platform system interface. To be WASI-compliant, each runtime must implement all the calls defined in the interface with dedicated functions, which are named hostcalls. However, implementing these functions is non-trivial, since (i) the code must not introduce violations to the Wasm memory model, and (ii) it is possible to break the separation between the system and the isolated environment in which the Wasm module is executed. The solution adopted by current runtimes leverages WASI Libc [27], a library providing POSIX-compatible APIs built on top of hostcalls.

Currently, every WASI-compliant runtime implements the proposed file system interface with a libpreopen-like layer [21]. Whenever the runtime receives a request to open a file, it first checks whether the path belongs to the authorized list of directories, then it opens the file on behalf of the Wasm program, redirecting the content to the caller. Previous work [7, 16, 18] proved the approach to be error-prone, leaving the system unprotected when a vulnerability was introduced in a hostcall wrapper (Figure 1). Moreover, this approach provides limited flexibility, as it is associated with directory-based granularity instead of file-based. Lastly, in order to audit the policy regulating resource access, one must find the permissions by looking at the code. We claim that there is no practical advantage in having several implementations of the same access control checks for different runtimes. Our idea is to replace the user-space runtime-specific security checks with a single in-kernel implementation that leverages eBPF [26]. There are considerable advantages in doing so: (i) it permits to decouple the implementation of hostcall wrappers and the access control details, minimizing the risk of bugs [3, 17, 25], (ii) it enables the introduction of per-module policies with file-based granularity, and (iii) it fulfills

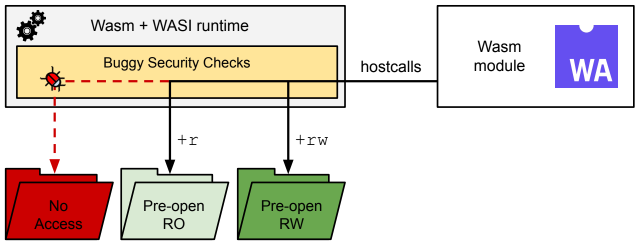

**Figure 1: Current implementation of WASI by runtimes. A bug present in a hostcall wrapper permits the module to read the unauthorized directory on the left (red dotted arrow)**

Wasm's promise of portability as eBPF programs are portable across different kernel versions [22] and also operating systems, thanks to Microsoft's undergoing effort to port eBPF to Windows [20].

## 2 THREAT MODEL

Our assumptions reflect the threat model employed by Wasm runtimes. We assume that the code executed by the runtime is either untrusted or it is trusted but potentially affected by security vulnerabilities due to bugs. The goal of the attacker providing the code is to bypass the security checks enforced by the runtime to get access to the host file system. To fulfill this objective, the attacker can leverage the interface provided by WASI and send any argument. Runtime escapes caused by memory corruption or alteration of the program flow are out of scope of our work, since protection can be provided by other existing solutions (e.g., [7]).

## 3 ARCHITECTURE

Our analysis starts from the scenario illustrated in Figure 1. Currently, WASI-compliant runtimes implement dedicated user-space wrappers to enforce the security boundaries of hostcalls. File system access is granted by the user on a set of pre-opened directories that are specified via CLI before the Wasm module is run (e.g., with the `--dir` option). We follow a similar approach, asking the user to state the permissions of each Wasm module in a JSON policy file. Contrary to existing runtimes, permissions can be granted with file-based granularity. Three permissions are available: (i) `read` to open and read a file, (ii) `write` to modify, truncate and append content to a file, and (iii) `delete` to remove the file. When permissions are related to a directory, `read` translates to listing its content, `write` allows to create and delete files within it. We extended the Wasmtime and WasmEdge runtimes to load the policy at startup, and, instead of pre-opening the directories available to the Wasm module, we enforce the policy with eBPF. eBPF code is split into programs attached to a kernel- or user-space function called *hook point* and executed whenever the hook is reached. Programs have visibility of function parameters, they can persist state and share it with user space using *maps*, and most of all they can enforce security checks based on this information. Once the policy is encoded inside the map and the eBPF programs are loaded, the runtime instantiates the Wasm module selected by the user (arrow Ⓐ in Figure 2). At this stage, the modified runtime invokes a dedicated user probe specifying as a parameter the policy that confines the loaded Wasm module (Ⓑ). The argument is captured by a dedicated eBPF program

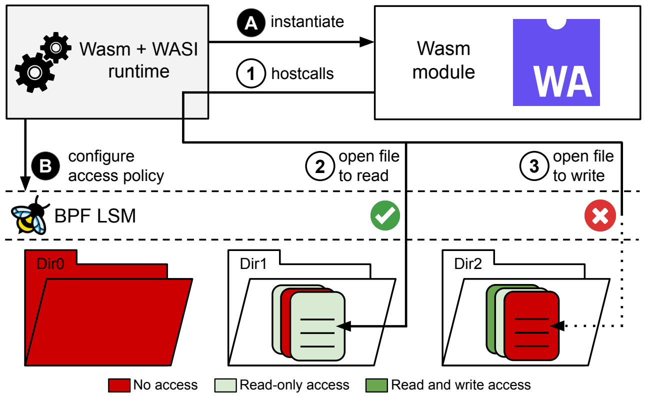

**Figure 2: Workflow of our proposal. The runtime instantiates the Wasm module (Ⓐ), and configures the associated policy calling the traced user probe (Ⓑ). After the Wasm module is run, all the hostcalls issued by the program (①) are restricted by eBPF (②, ③)**

that also annotates the identifier of the thread running the Wasm interpreter in a tracing map. We highlight that the policy is activated before the runtime executes the module (i.e., before untrusted code is interpreted). The consequence is that, from this point on, all the hostcalls performed by the Wasm module are restricted by our eBPF programs (arrows ①, ②). The eBPF programs that make the security decisions are evaluated every time a file-related kernel security hook is reached (e.g., `security_file_open`), and any access decision is enforced at kernel level. When an unauthorized request is performed by the Wasm code (③), the related eBPF program detects the violation and denies the request, returning to the caller a permission denied error. When the execution of untrusted Wasm code terminates, another eBPF program is responsible for removing the access restriction from the thread executing the Wasm runtime. No further intervention from the runtime is required, as the maps and the eBPF programs are automatically removed from the kernel immediately after the process running the runtime terminates.

This architecture offers several advantages. First, it eliminates the risks coming from buggy user-space security checks (e.g., wrong filepath resolution [19], wrong directory removal [8]). Then, by leveraging kernel hook points [26], our approach allows the runtime developer to focus on the interaction between Wasm code and the memory unsafe system call, leaving aside authorizations and policy-related issues. Lastly, access constraints can be audited by simply looking at the JSON policy, instead of inspecting the code.

## 4 EXPERIMENTS

To investigate the overhead introduced by our solution we implemented it in WasmEdge and Wasmtime, two industrial state-of-the-art Wasm runtimes. The evaluation has been performed in the following test environment: an Ubuntu 22.04 LTS server powered by an AMD Ryzen 2950X CPU with 16 cores, 128 GB RAM, and 2 TB SSD. In order to assess the performance, we tested one of the most popular binaries that can be compiled to Wasm with support to WASI: `uutils coreutils`, the porting of the coreutils in Rust [9]. First, we compiled the coreutils with the `wasm32-wasi` target, and applied runtime-specific optimization (with `wasmedgec` [2]

| Utility | WasmEdge | WasmEdge* | Wasmtime | Wasmtime* |
|---|---|---|---|---|
| `head` | 32 | 34 (+6.25%) | 14 | 16 (+14.29%) |
| `sum` | 134 | 137 (+2.24%) | 130 | 136 (+4.62%) |
| `tac` | 149 | 150 (+0.67%) | 152 | 155 (+1.97%) |
| `wc` | 285 | 287 (+0.70%) | 309 | 310 (+0.32%) |
| `shuf` | 298 | 300 (+0.67%) | 356 | 358 (+0.56%) |
| `ls` | 512 | 526 (+2.73%) | 1077 | 1113 (+3.34%) |
| `seq` | 1155 | 1157 (+0.17%) | 1526 | 1533 (+0.46%) |
| `cut` | 1403 | 1411 (+0.57%) | 359 | 360 (+0.28%) |
| `join` | 1601 | 1603 (+0.12%) | 2054 | 2065 (+0.54%) |
| `split` | 4416 | 4694 (+6.30%) | 4933 | 4998 (+1.32%) |

**Table 1: Average execution time in *ms* of the coreutils without and with* our approach (% overhead in parenthesis)**

for WasmEdge and with `wasmtime compile` [1] for Wasmtime) to further speed up the code. Then, we reproduced the benchmarks reported in the coreutils repository, with the exception of those that are not portable to WASI due to temporary lack of support (e.g., the `dd` utility needs to spawn threads, a feature that is yet to be implemented [11]). Finally, we repeated the experiments with our protection in place. The Hyperfine benchmarking tool [10] was used to log measures, and 1000 runs were performed (with 100 warmups). As shown from the results in Table 1, our approach introduces a limited overhead, ranging from an additional 0.12% to 6.30% for WasmEdge, and from 0.28% to 14.29% for Wasmtime. As expected, the highest overhead is experienced by short-living utilities (e.g., `head`). We also observe that there are notable differences between the WasmEdge and Wasmtime test execution time for some utilities (e.g., `ls` and `cut`); from our analysis these differences are mostly caused by the specific post-compilation optimizations.

## 5 RELATED WORK

There are several successful solutions that leverage Wasm to sandbox untrusted code [14, 23, 24]. RLBox [23] is a framework that facilitates the isolation of third-party libraries in pre-existing software. eWASM [24] optimizes the execution of Wasm in embedded systems with constrained resources. Sledge [14] enables efficient Wasm-based serverless execution on the edge. The use of our approach for restricting access to the file system within these frameworks can strengthen their security assurance.

The memory safety guarantees of Wasm depend on the runtime implementation [18]. Hence, Bosamiya et al. [7] explore the problem of producing provably safe sandboxes. WaVe [16] explains that any interaction with the unsafe interfaces exposed by WASI can introduce security and safety violations. Thus, the authors proposed a verified secure runtime system implementing WASI. However, both works require to redesign the runtime toolchain, while our solution can be directly integrated into existing runtimes.

The academic and industrial communities have investigated the use of eBPF for the isolation of software [4, 5, 12, 13]. *BPFBox* [13] and *BPFContain* [12] use an eBPF daemon to confine processes and services. *Cilium* [4] provides eBPF-based networking, observability and security for container workloads. *Falco* [5] enables lightweight threat detection in the cluster. These solutions highlight the potential of eBPF, and provide a simple and flexible confinement of system resources. However, they focus on containers or services, while our solution aims at enforcing fine-grained per-sandbox policies.

## 6 CONCLUSIONS

The results achieved by our approach are promising: not only it permits to introduce fine-grained policies to restrict file system access, it is also associated with a limited overhead which is aligned with the needs of a modern sandbox. The protection is currently applied only to the file system, but our approach has the potential to be extended also to network sockets, which are in the first stage of the standardization process [6]. We believe this could be an interesting line of research for future work.

## ACKNOWLEDGMENTS

We thank the reviewers for their valuable comments and feedback. This work was supported by the European Commission in the HE program within the GLACIATION project (No 101070141).

## REFERENCES

[1] 2023. CLI Options - Wasmtime. https://docs.wasmtime.dev/cli-options.html
[2] 2023. wasmedgec AOT Compiler - WasmEdge. https://wasmedge.org/book/en/cli/wasmedgec.html
[3] M. Abbadini, D. Facchinetti, G. Oldani, M. Rossi, and S. Paraboschi. 2023. Cage4Deno: A Fine-Grained Sandbox for Deno Subprocesses. In *ASIACCS*.
[4] The Cilium Authors. 2023. Cilium. https://cilium.io
[5] The Falco Authors. 2022. Falco. https://falco.org
[6] D. Bakker. 2023. wasi-sockets. https://github.com/WebAssembly/wasi-sockets
[7] J. Bosamiya, W. S. Lim, and B. Parno. 2022. Provably-Safe Multilingual Software Sandboxing using WebAssembly. In *USENIX Security*.
[8] B. Coenen. 2021. feat(wasi): add rename for a directory + fix remove_dir. https://github.com/wasmerio/wasmer/commit/e0e12f9d9ff41a512e44bd497324e
[9] The coreutils Authors. 2023. uutils coreutils. https://github.com/uutils/coreutils
[10] P. David. 2023. *hyperfine*. https://github.com/sharkdp/hyperfine
[11] A. Ene, M. Kolny, and A. Brown. 2023. wasi-threads. https://github.com/WebAssembly/wasi-threads
[12] W. Findlay, D. Barrera, and A. Somayaji. 2021. BPFContain: Fixing the Soft Underbelly of Container Security. *arXiv* (2021).
[13] W. Findlay, A. Somayaji, and D. Barrera. 2020. bpfbox: Simple Precise Process Confinement with eBPF. In *Cloud Computing Security Workshop*.
[14] P. K. Gadepalli, S. McBride, G. Peach, L. Cherkasova, and G. Parmer. 2020. Sledge: A Serverless-First, Light-Weight Wasm Runtime for the Edge. In *International Middleware Conference*.
[15] A. Haas, A. Rossberg, D. L. Schuff, B. L. Titzer, M. Holman, D. Gohman, L. Wagner, A. Zakai, and JF Bastien. 2017. Bringing the web up to speed with WebAssembly. In *Programming Language Design and Implementation*.
[16] E. Johnson, E. Laufer, Z. Zhao, D. Gohman, S. Narayan, S. Savage, D. Stefan, and F. Brown. 2022. WaVe: A Verifiably Secure WebAssembly Sandboxing Runtime. In *IEEE Security and Privacy*.
[17] M. Kehoe. 2022. eBPF: The Next Power Tool of SREs. USENIX Association.
[18] D. Lehmann, J. Kinder, and M. Pradel. 2020. Everything old is new again: Binary security of webassembly. In *USENIX Security*.
[19] M. McCaskey. 2019. Prevent parent directory from being opened without being preopened wasi. https://github.com/wasmerio/wasmer/pull/463
[20] Microsoft. 2023. eBPF for Windows. https://microsoft.github.io/ebpf-for-windows/
[21] MUSEC. 2023. libpreopen. https://github.com/musec/libpreopen
[22] A. Nakryiko. 2021. BPF CO-RE. https://nakryiko.com/posts/bpf-core-reference-guide/
[23] S. Narayan, C. Disselkoen, T. Garfinkel, N. Froyd, E. Rahm, S. Lerner, H. Shacham, and D. Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *USENIX Security*.
[24] G. Peach, R. Pan, Z. Wu, G. Parmer, C. Haster, and L. Cherkasova. 2020. eWASM: Practical Software Fault Isolation for Reliable Embedded Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2020).
[25] M. Rossi, D. Facchinetti, E. Bacis, M. Rosa, and S. Paraboschi. 2021. SEApp: Bringing Mandatory Access Control to Android Apps. In *USENIX Security*.
[26] The kernel development community. 2023. LSM eBPF Programs. https://docs.kernel.org/bpf/prog_lsm.html
[27] WebAssembly. 2023. WASI Libc. https://github.com/WebAssembly/wasi-libc
[28] WebAssembly. 2023. The WebAssembly System Interface. https://wasi.dev