Comparing Security in eBPF and WebAssembly

Jules Dejaeghere*
University of Namur
Belgium
jules.dejaeghere@unamur.be

Bolaji Gbadamosi*
Karlstad University
Sweden
bolaji.gbadamosi@kau.se

Tobias Pulls
Karlstad University
Sweden
tobias.pulls@kau.se

Florentin Rochet
University of Namur
Belgium
florentin.rochet@unamur.be

ABSTRACT

This paper examines the security of eBPF and WebAssembly (Wasm), two technologies that have gained widespread adoption in recent years, despite being designed for very different use cases and environments. While eBPF is a technology primarily used within operating system kernels such as Linux, Wasm is a binary instruction format designed for a stack-based virtual machine with use cases extending beyond the web. Recognizing the growth and expanding ambitions of eBPF, Wasm may provide instructive insights, given its design around securely executing arbitrary untrusted programs in complex and hostile environments such as web browsers and clouds. We analyze the security goals, community evolution, memory models, and execution models of both technologies, and conduct a comparative security assessment, exploring memory safety, control flow integrity, API access, and side-channels. Our results show that eBPF has a history of focusing on performance first and security second, while Wasm puts more emphasis on security at the cost of some runtime overheads. Considering language-based restrictions for eBPF and a security model for API access are fruitful directions for future work.

CCS CONCEPTS

• **Security and privacy → Operating systems security**; **Browser security**; • **Software and its engineering → Software design tradeoffs**.

KEYWORDS

eBPF, WebAssembly, Security Comparison, Threat Model, Memory Safety, Control Flow Integrity, API Access, Side-channels

ACM Reference Format:
Jules Dejaeghere, Bolaji Gbadamosi, Tobias Pulls, and Florentin Rochet. 2023. Comparing Security in eBPF and WebAssembly. In *Workshop on eBPF and Kernel Extensions (SIGCOMM '23), September 10, 2023, New York, NY, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3609021.3609306

*Shared first author.

eBPF '23, September 10, 2023, New York, NY, USA

ACM ISBN 979-8-4007-0293-8/23/09.
https://doi.org/10.1145/3609021.3609306

1 INTRODUCTION

About a decade ago, eBPF got merged into the Linux kernel as a technology for running custom eBPF programs at specific hook points within the privileged context of the kernel without directly modifying the kernel [16]. In gist, it made the kernel dynamically programmable. Since then, eBPF has seen wide adoption, ranging from support in multiple operating systems (e.g., Windows [42] and FreeBSD [25]), new use cases (e.g., XDP [28] and tracing [10]), and broader support beyond operating system kernels (e.g., eBPF in hardware [26] and userspace [46]). Recently, due to the scope and importance of eBPF growing, the IETF has started a working group on standardizing the core of eBPF [30].

In parallel with the developments of eBPF, all major web browsers have shipped support for WebAssembly (Wasm) [24]. Wasm is a binary instruction format designed for a stack-based virtual machine, intended to serve as a portable target for high-level languages like C and Rust. With its efficient execution speed, compact binary format, and a high degree of security, Wasm has gained widespread use on the web. Much like eBPF, Wasm has also expanded beyond the confines of the web, with various use cases including edge computing (e.g., Fastly's Compute@Edge [18]), serverless computing (e.g., Cloudflare Workers [12]), and standalone applications (e.g., Wasmer runtime [56] and Wasmtime [5]).

In this paper, we *compare eBPF and Wasm, focusing on the security* of the technologies (as opposed to how to use the technologies for security purposes, e.g., Cilium [11]). As the scope and ambitions of eBPF grow, our starting point is that Wasm may provide valuable lessons, as Wasm was designed around securely and safely running untrusted programs in web browsers [24]: a highly complex and hostile environment on par with operating system kernels.

While designed for widely different use cases and environments, the lifecycle of eBPF and Wasm programs have many similarities. Figure 1 shows an overview of a *possible* lifecycle for both technologies, starting from source code in some supported language (e.g., C, C++, or Rust) that is compiled (e.g., using LLVM) into either eBPF or Wasm. Next, the program is verified as safe (eBPF) or placed in a locked-down runtime (Wasm) prior to execution within a trust boundary (light grey background). Programs then execute in the respective environments, e.g., the Linux kernel (eBPF) or a runtime (Wasm). The program is either isolated from the host (Wasm) or determined not to harm the kernel (eBPF). Any system interaction outside the respective isolated environments is through dedicated

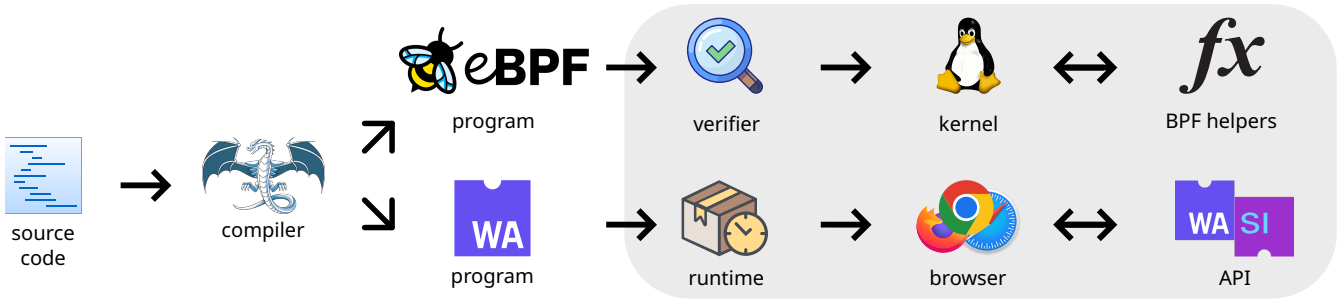

Figure 1: An overview of a possible lifecycle of eBPF and Wasm programs: from source code to execution of potentially untrusted programs within some trust boundary (shaded area).

APIs, e.g., through allow-list functions called "helpers" for eBPF in the kernel or WASI (WebAssembly System Interface) for Wasm.

The structure of the remainder of the paper is as follows: Section 2 surveys related work on the security of eBPF and Wasm. Sections 3 and 4 dissect eBPF and Wasm, respectively, scrutinizing their main security goals, community evolution, memory models, and execution models. Section 5 performs a comparative security analysis of eBPF and Wasm, examining memory safety, control flow integrity, API access, and side-channel vulnerabilities. It also juxtaposes the security approaches of eBPF and Wasm relating to potential vulnerabilities like buffer overflows and infinite loops, explained through code examples. Section 6 presents key findings and suggests directions for future work. In particular, Wasm's emphasis on security led to language-based restrictions and a capability-based security model for API access. Considering such changes for eBPF—with a focus on minimizing the potential negative impact on performance— are fruitful directions for future work.

2 RELATED WORK

Several solutions use either eBPF or Wasm to sandbox untrusted code [5, 11, 12, 15, 17–21, 43, 52, 56, 62]. In addition, some frameworks combine the best from both technologies to offer even more efficient solutions (cf. [1]). However, more research must be conducted to compare the security aspects between eBPF and Wasm.

Huang and Paradies [29] qualitatively and quantitatively evaluated eBPF and Wasm as offloading mechanisms in the context of computational storage. They evaluated compatibility, ergonomics, language agnosticism, portability, tooling, and safety. In summary, they find that Wasm is favourable across the board, except for ergonomics (a tie) and safety. In the context of safety, they rate Wasm as good and eBPF as unknown. By unknown, they mean that the dependency on the Linux kernel verifier for safety could be more practical in their context of computational storage. They highlight that they only considered safety, and not security, by defining safety as protection against human errors and security as protection against attacks. In this paper, we fill this gap by comparing the security and safety aspects of eBPF and Wasm.

2.1 eBPF security

Kirzner and Morrison [35] showed how unprivileged eBPF and a verifier vulnerability could be used to create a Spectre universal read gadget in Linux. Vishwanathan et al. [54] provided formal proof of correctness and optimality of addition, subtraction, and multiplication operations in the verifier. Their research reinforces the accuracy and dependability of eBPF programs, enhancing their overall security. Gershuni et al. [22] designed an advanced static analyzer for eBPF using abstract interpretation, named PREVAIL, the default verifier in the Windows eBPF ecosystem. This analyzer provides enhanced precision and a broader scope, enabling the verification of eBPF programs that include loops. Importantly, their approach maintains safety while addressing complex control flow in eBPF programs. Nelson et al. [45] applied formal methods to BPF just-in-time compilers in the Linux kernel, providing a verified optimization step *after* a program is deemed safe by the verifier. Lu et al. [38] suggest isolating the execution of BPF programs post-verification as a hardening mechanism using the emerging Intel Memory Protection Keys hardware extension. Part of this entails categorizing the 200+ existing BPF helper functions based on their type of access to kernel memory.

2.2 WebAssembly security

We distinguish between binary and host security [37].

Binary security concerns the security of the code running inside a Wasm runtime. Vulnerabilities can propagate from unsafe languages being compiled to Wasm and may even pose a more significant threat due to the lack of exploit mitigation techniques (e.g., stack canaries, ASRL) [6, 27, 37, 41].

Host security concerns the security of the host system executing the Wasm code with a runtime. Bosamiya et al. [9] classify existing runtimes regarding security and performance trade-offs and introduce two high-performance sandboxing compilers using F* and Rust. Fundamentally, APIs exposed to the sandboxed code in the runtime are not within the scope of any security or safety guarantees, e.g., file access provided through WebAssembly System Interface (WASI) [23] may threaten host security. Johnson et al. [33] present WaVe, a verifiably secure WASI runtime. Note that WASI has apparent security goals that make this possible. Jangda et al. [31] identify safety checks that are inherent to Wasm's safety guarantees: stack overflow checks, indirect call checks, and reserved registers, which all contribute to increased code size and runtime overheads in comparison to native code.

3 EBPF

3.1 Main goals

Initially, eBPF programs, modeled around untrusted, unprivileged BPF, had limited kernel capabilities and aimed to make this approach safe and secure. However, due to the overwhelming complexity of Linux, most distributions today require root or CAP_BPF privileges to run eBPF programs [14]. What remains is that the main goal is to protect kernel integrity through the use of the eBPF verifier [51]. Programs deemed safe for the kernel can call BPF helpers—kernel functions eBPF programs use to interact with the kernel, perform operations, and access kernel data structures.

3.2 Community and evolution

The development and governance of eBPF, which primarily focuses on the Linux kernel development process, is overseen by the Linux community and its maintainers.

Beyond its original purpose of network packet filtering, eBPF has evolved through the contributions of a thriving open-source community. It is now used for various applications, including performance monitoring, security monitoring, network analysis, and more [39]. The community remains committed to advancing its development, expanding its capabilities, and ensuring its value to developers and system administrators in the Linux ecosystem.

The eBPF development process entails the submission of patches and enhancements to the eBPF subsystem inside the Linux kernel via a designated mailing list. The community and maintainers review the code, provide feedback, and ensure that the modifications are in line with the Linux kernel's goals and principles [34].

3.3 Memory model

In eBPF, the concept of memory refers to the memory space utilized by eBPF programs for storing and manipulating data. eBPF does not have its specific memory model but instead relies on the Linux kernel memory model, at best [13].

Local variables are stored in stack memory. This stack memory is a continuous block of memory, and the stack pointer is responsible for keeping track of the current position within the stack. Memory in eBPF is restricted due to a maximum stack of 512 bytes and not allowing heap allocations. To overcome these constraints, eBPF programs use BPF maps to facilitate communication and data sharing between eBPF programs and user processes. BPF maps exist in various types and sizes, allowing for various data storage and retrieval processes [48]. They can be accessed by several eBPF programs and user processes at the same time, enabling efficient communication and data sharing [44]. It is essential to understand that eBPF programs operate within a controlled memory environment in the kernel and are not granted arbitrary access to system memory. The eBPF verifier is crucial in ensuring memory safety by scrutinizing eBPF programs for potential issues such as accessing memory beyond its bounds or reading uninitialized memory. The verifier keeps track of the memory each register points to during simulated execution and prohibits unsafe access. This verification process significantly mitigates security risks and memory access violations, thus promoting the overall safety of the kernel.

3.4 Execution model

The abstract eBPF virtual machine provides a lightweight execution environment with 10 general-purpose registers and a read-only frame pointer dedicated to stack access [49]. eBPF programs consist of one or more subprograms (functions) of bytecode instructions. When eBPF programs are compiled from source code, the compiler also produces BTF (BPF Type Format) debug information of the program, which is used by the verifier together with existing BTF information about the kernel [50].

Programs are *loaded* into the kernel with the BPF_PROG_ LOAD system call, which invokes the eBPF verifier to ensure the program's safety. The verified program is typically JIT compiled if deemed safe, and a file descriptor is returned to user space. The user may then *attach* the program using BPF_PROG_ATTACH to predefined hooks in the kernel that give programs direct access to system calls, kernel functions, and network events. Programs are executed by the kernel when relevant events occur within the kernel.

Fundamentally, the eBPF instruction set is permissive: all security and safety of programs stem from using the eBPF verifier [49, 51]. The eBPF verifier ensures the safety of eBPF programs through a multi-phase approach. In the parsing phase, the bytecode is checked for syntactic correctness. In the control flow verification phase, the program's control flow graph is analyzed to prevent unbounded loops, invalid code, and ensure proper termination. This is done by exhaustively analyzing every possible path through the program—potentially falsely rejecting safe programs due to state explosion. In the type-checking phase, the verifier enforces the compatibility of variables and expressions with their types using BTF information and prevents type mismatches, invalid memory accesses, and buffer overflows. In the data flow analysis phase, the verifier tracks register and stack state to enforce bound checks, prevent invalid operations, and validate helper function usage. The verifier also ensures safe program execution by preventing access to uninitialized or out-of-bounds memory, checking for NULL values before dereferencing and validating the use of spin locks to prevent deadlocks and ensure synchronization.

4 WEBASSEMBLY

4.1 Main goals

Wasm is a portable low-level bytecode. It aims to enable safe, fast, portable, and compact programs on the web [24, 60]. However, Wasm does not make any web-specific assumptions, so it can be deployed in other environments as well [60]. Wasm comes with two main goals regarding security: protecting users from buggy or malicious code and providing developers with useful primitives for developing safe applications [59].

Those main goals of Wasm motivated choices towards hardware, language, and platform independence. Indeed, Wasm is a portable compilation target that can eventually be compiled once more to native opcodes. Additionally, the choice is left to the developer regarding the platform on which Wasm is compiled: web browser, stand-alone virtual machine, or integrated into other environments [60].

4.2 Community and evolution

Wasm is defined in multiple specification documents. The core specification [60] defines the semantics of the modules in one document, independently from any embedding. This independence between the specification of the language and its embedding enables hardware and platform independence. In addition to this core specification, other specifications exist for APIs. The interaction with the host system has its own specification.

New features to Wasm or specified APIs are submitted via feature proposals. Such proposals go through a multiple-step process leading to the eventual standardization of the features [61].

Aside from the specifications of Wasm and the APIs, the Bytecode Alliance [2]—a non-profit organization—focuses on creating a shared implementation of the standards presented in the specifications. Those shared implementations enable industrial actors, researchers, and individuals to experience Wasm and provide feedback to inform the standardization process. Current projects driven by the Bytecode Alliance include Wasmtime [5] (a standard compliant Wasm runtime), Cranelift [3] (the code generator used by the just-in-time compiler of Wasmtime), and WAMR [4] (a lightweight runtime for Wasm).

4.3 Memory model

The main memory of a Wasm program is an array of bytes, called *linear memory*. This linear memory is disjoint from code space, execution stack, and runtime data structure [24], preventing Wasm modules from accessing other memory areas than the linear memory. The execution stack mainly stores local variables, global variables, and return addresses. Modules can access the data stored on the execution stack via dedicated instructions. The actual data address on the execution stack is never shown to the module.

As the execution stack may only hold data of one of the four primitive types defined by Wasm, compilers targeting Wasm implement their own stack, residing in the linear memory. This enables the program to store non-scalar data and any variable whose address needs to be taken by the module [37].

Compilers targeting Wasm also create an area for the heap in the linear memory. This area is reserved at the end of the linear memory so it can dynamically grow when additional space is allocated for the linear memory.

Access to the linear memory by the module is dynamically bound-checked, preventing modules from accessing data outside their allocated memory [24]. However, bound checking is performed at the level of the linear memory: modules can access the entire linear memory of the module without restriction. Linear memory is not protected by standard techniques like stack canaries [55] or guard pages.

4.4 Execution model

Wasm code is executed when instantiating a module or when an exported function is invoked on a given instance [60]. The execution behavior of a Wasm module is defined in terms of an abstract machine that models the program state. This abstract machine includes a stack (recording the operand values and control constructs) and an abstract store (containing the global state).

A compliant runtime ensures that the module does not break Wasm's memory model [60]. This is done by bound checking the access to the linear memory: if the module accesses the memory outside of the linear memory, the program traps.

The definition of the Wasm bytecode [60] limits the constructs that are possible to express. Arbitrary jumps (e.g., go-to statements) are not allowed, only structured control flow is provided by Wasm. Consequently, a grammatically valid Wasm module can only jump to the beginning of valid constructs (e.g., conditional constructs or functions) [24].

Similarly, restrictions are applied regarding functions that the module can indirectly call. To indirectly call a function, the module provides a runtime index to a table. This table holds the signatures of the functions that the module defines or imports and that can be indirectly called. When an indirect call is done, the runtime checks that the calling signature and the signature of the called function match. In case of a type mismatch or an out-of-bounds table access, a trap occurs [24].

5 COMPARISON

To supplement the comparison, we summarize results in Table 1 and provide textbook code examples in the following gist: https://gist.github.com/Rekindle2023/ca6a072205698925fa80f928eebe172e.

5.1 Threat model

While eBPF as a language lacks a threat model, eBPF in the context of the Linux kernel is focused on kernel integrity with the help of the eBPF verifier acting as a gatekeeper. Executing programs is typically a privileged system operation (i.e., root or CAP_BPF). Despite excessive checks and limitations imposed by the verifier, the developments around eBPF have essentially given up on untrusted programs [14].

Wasm focuses on enabling untrusted code to run on a system without being able to compromise the host. This threat model is in line with one of the first goals of Wasm: bringing a portable bytecode format to the web, where payloads are downloaded from sources that are often untrusted. To this end, Wasm limits the actions that a module can take on the system regarding memory access (see Section 4.3) and branching (see Section 4.4). By following the specifications [60], the host's runtime has to implement runtime checks that ensure the isolation of the modules.

5.2 Memory safety

Both eBPF and Wasm aim to provide memory safety during execution. However, their respective approaches are fundamentally different.

eBPF enables programmers to write programs with few limitations regarding the constructs they can use. Some programs written in eBPF are unsafe to execute and result in corrupted kernel memory. For this reason, the eBPF verifier is in charge of inspecting the eBPF programs and ensuring that they are safe to execute. If the verifier cannot prove safety, the program will not be executed in the kernel.

On the other hand, the Wasm bytecode specification [60] only provides a limited set of constructs (see Section 4.4). All grammatically correct programs in Wasm can be executed in a runtime.

Table 1: Comparison results between eBPF in the Linux kernel and WebAssembly in Wasmtime.

Feature	Linux Kernel eBPF	Wasmtime and WebAssembly
Threat model	– Kernel integrity and safety – Trusted programs	– Untrusted code – Distinguish binary and host security
Memory safety	– Type check – Memory bound check – Pointer and stack bound check	– Sandbox with runtime checks – Managed stack – Traps
Control flow integrity	– Bounded loops and instructions (1 million) – Checks and rejects unreachable instructions – Symbolic execution jump verification – Ensures program termination	– Type checking – Return address on the managed stack – Structured control flow only – Jump only to start of constructs
API access	– BPF helpers – Disable unprivileged BPF	– API access provided by the host – WASI capability-based security model
Side-channels	– Constant blinding – Impossible path verification – Retpolines	– Wasm language is potentially vulnerable, not in scope of specification – Wasmtime started to implement mitigations

Due to the runtime checks, Wasm modules are constrained in their memory accesses and indirect function calls (see Sections 4.3 and 4.4).

To illustrate how eBPF and Wasm deal with memory safety, we implemented C programs with textbook examples of buffer overflow (misusing `strcpy()`) and reading outside the allocated buffer. The eBPF verifier halted on both examples to prevent their execution. Wasm let both examples run to completion: the programs were compiled to valid Wasm bytecode and did not exceed the bounds of their linear memory (see Section 4.3).

These observations raise pertinent questions regarding the balance between flexibility and safety. Is the trade-off of greater programming freedom in eBPF worth the additional verification step? Conversely, does the constrained nature of Wasm compromise developers' capabilities, or does it offer stronger memory safety guarantees?

5.3 Control flow integrity

eBPF and Wasm have mechanisms to enforce control flow integrity but handle it in different ways. In eBPF, the control flow integrity is primarily enforced by the verifier, which analyzes the control flow structure and ensures it conforms to established rules (see Section 3.4). If any violations are detected, the verifier flags them as errors and prevents the program from being loaded into the kernel.

Wasm, on the other hand, primarily achieves control flow integrity through the execution semantics of the language itself. As mentioned in Section 4.4, Wasm defines valid code constructs and how control flow may only jump to the beginning of a valid construct. Indirect function calls (also described in Section 4.4) prevent call redirection in Wasm.

As a simple illustration of the control flow integrity strategies of eBPF and Wasm, we wrote an infinite C loop. The program compiled to eBPF was rejected by the verifier and could not be loaded in the kernel. However, Wasmtime executed the program and hanged in the loop: this does not violate the memory safety of the host, while some runtimes implement controls (e.g., Wasmtime [5, 57]) to prevent programs from consuming too many CPU cycles.

5.4 API access

By default, Wasm does not have access to the resources of the host (e.g. file system, network, system calls). Modules can import externally defined functions provided by the host or other modules. APIs common to many use cases are currently being standardized in the WebAssembly System Interface (WASI) [23]. The capability-based security model of WASI enabled Johnson et al. [33] to introduce a verified secure runtime system that implements WASI.

On the contrary, eBPF programs within the Linux kernel have access to over 200 helper functions by default [47]. Certain helper functions, such as `bpf_probe_write_user()` (write bytes to a user-space memory address) and `bpf_override_return()` (override the return value of a function), can be detrimental to system security, allowing manipulation of user space memory, overriding return values, dropping packets, and trigger reads and writes to kernel memory that may leak information or put the overarching system into an insecure state [7, 38]. Access to helpers is restricted based on eBPF program type and if unprivileged BPF is enabled [32].

5.5 Side-channels

eBPF uses various security measures to mitigate side-channel attacks, including Spectre-like attacks [36]. The verifier incorporates measures such as constant blinding and enforcing speculation bounds [8, 53]. It also analyzes the control flow of the eBPF program, validates impossible paths, and uses retpolines to direct the flow of execution predictably. It also inserts `nospec` instructions

as a speculation barrier, mapped to potential CPU architecture mitigations.

The Wasm language specification [60] clearly states that side-channel attacks are to be addressed by the runtime. Currently, Wasmtime implements a few forms of Spectre mitigations. Bounds checks for the runtime index used in indirect calls are mitigated. Some other instructions are mitigated to ensure that speculation goes to a deterministic place [58]. New Spectre mitigations will be added to Wasmtime as it still is a subject of ongoing research.

6 CONCLUSIONS

Both eBPF and Wasm, with their respective technology stacks, aim to provide secure, safe, and fast execution of a portable bytecode format. However, their approaches on how to enforce this are fundamentally different.

While the eBPF bytecode leaves the programmer with much freedom, Wasm provides a limited number of constructs to the programmer. The eBPF verifier is the designated gatekeeper in charge of flagging unsafe programs and preventing them from being loaded in the kernel. As the verifier uses a heuristic technique to verify programs conservatively, some safe programs may be rejected, and unforeseen unsafe constructs might be accepted. WebAssembly runtimes, on the other hand, can rely on the bytecode specification to determine what actions a module may perform. Runtimes then have to dynamically check some designated operations the modules take to ensure safety and security at a performance cost compared to native code.

Both eBPF and Wasm started with different threat models that reflect their original design goals. While eBPF assumes mainly trusted software, Wasm enables the execution of untrusted code. These threat models influenced the technologies' approaches to addressing issues such as memory safety, control flow integrity, and access to host functions.

eBPF in the Linux kernel delegates the memory safety and control flow integrity to the verifier and provides a wide range of helper functions to the programs to call host functions. Wasm performs memory safety checks at runtime and uses type checking, structured control flow, and a managed stack to provide control flow integrity by design. The host provides the Wasm modules only with the necessary host functions.

eBPF programs in the Linux kernel can access a wide range of helper functions by default, enabling greater system interaction. However, this raises security concerns as certain helper functions can be misused, potentially compromising system integrity. Wasm modules do not have direct access to host resources by default. Instead, controlled and secure access is enabled through externally defined functions and standardized APIs like the WebAssembly System Interface (WASI). To further advance the research, future investigations could explore ways to leverage the runtime checks and security measures employed by Wasm to enhance eBPF, providing valuable insights into concrete strategies for improving security and memory safety while minimizing performance overhead.

Overall, eBPF has a history of focusing on performance first and then implementing security. Wasm does the opposite by first defining the security property via sandboxing and isolation and then minimizing the overhead. Depending on the use case, similar overheads as in Wasm may not be acceptable in eBPF programs that are meant to run in the kernel. eBPF enables native execution by accepting the limitations introduced by the verifier.

Future research could focus on quantifying the performance differences between eBPF and Wasm. Assessing performance in light of the respective security measures would provide valuable insight into the actual performance cost of the additional security considerations in Wasm. With a better understanding of the performance differences, another research direction would be to investigate if the eBPF language itself could be redesigned to improve security—perhaps with inspiration from Wasm—while minimizing any negative impact on performance. Finally, BPF helpers could likely benefit from a security model to enable users to reason about their effects on *system* security, regardless of if unprivileged BPF remains disabled or not [14, 40]. As eBPF and WebAssembly are being used in an increasing number of applications, a quantitative comparison of their performance may help future researchers and developers to choose the most appropriate technology. Similarly, highlighting their differences and respective strengths in a deeper analysis may guide future design decisions.

ETHICS STATEMENT

Our research aims to improve the security of the eBPF or Wasm ecosystems by means of analytical comparison. The only artifact of our research is small code snippets of common vulnerabilities. This research did not risk causing harm and therefore did not raise ethical concerns.

ACKNOWLEDGEMENTS

This research was partially funded by the CyberExcellence project of the Public Service of Wallonia (SPW Recherche), convention No. 2110186, and Red Hat Research in the https://research.redhat.com/blog/research_project/security-and-safety-of-linux-systems-in-a-bpf-powered-hybrid-user-space-kernel-world/ project.

REFERENCES

[1] 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)* (Melbourne, VIC, Australia). https://doi.org/10.1145/3579856.3592831

[2] Bytecode Alliance. 2023. Bytecode Alliance. https://bytecodealliance.org/ accessed 2023-06-08.

[3] Bytecode Alliance. 2023. Cranelift. https://cranelift.dev/ accessed 2023-06-09.

[4] Bytecode Alliance. 2023. WAMR: WebAssembly Micro Runtime. https://wamr.dev/ accessed 2023-06-09.

[5] Bytecode Alliance. 2023. Wasmtime: A fast and secure runtime for WebAssembly. https://wasmtime.dev/ accessed 2023-06-09.

[6] John Bergbom. 2018. Memory safety: old vulnerabilities become new with WebAssembly. Technical report, Forcepoint.

[7] Dave Bogle. 2023. eBPF: A new frontier for malware. https://redcanary.com/blog/ebpf-malware/ accessed 2023-05-22.

[8] Daniel Borkmann. 2023. BPF and Spectre: Mitigating transient execution attacks – Daniel Borkmann, Isovalent. https://www.youtube.com/watch?v=6N30Yp5f9c4 accessed 2023-06-09.

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

[10] Cyril Renaud Cassagnes, Lucian Trestioreanu, Clément Joly, and Radu State. 2020. The rise of eBPF for non-intrusive performance monitoring. In *NOMS 2020 - IEEE/IFIP Network Operations and Management Symposium, Budapest, Hungary, April 20-24, 2020*.

[11] Cilium. 2023. Cilium - Cloud Native, eBPF-based Networking, Observability, and Security. https://cilium.io/ accessed 2023-06-09.

[12] CloudFlare. 2023. CloudFlare Docs: WebAssembly (Wasm). https://developers.cloudflare.com/workers/platform/webassembly/ accessed 2023-06-09.

[13] Jonathan Corbet. 2019. Concurrency management in BPF. https://lwn.net/Articles/779120/

[14] Jonathan Corbet. 2019. Reconsidering unprivileged BPF [LWN.net]. https://lwn.net/Articles/796328/ accessed 2023-06-09.

[15] Quentin De Coninck, François Michel, Maxime Piraux, Florentin Rochet, Thomas Given-Wilson, Axel Legay, Olivier Pereira, and Olivier Bonaventure. 2019. Pluginizing QUIC. In *Proceedings of the ACM Special Interest Group on Data Communication* (Beijing, China) *(SIGCOMM '19)*. Association for Computing Machinery, New York, NY, USA, 59–74. https://doi.org/10.1145/3341302.3342078

[16] eBPF.io authors. 2022. eBPF Documentation: What is eBPF? https://ebpf.io/what-is-ebpf/ accessed 2023-05-22.

[17] Authors Falco. 2023. Falco. https://falco.org.

[18] Fastly. 2023. Fastly Compute@Edge. https://docs.fastly.com/products/compute-at-edge accessed 2023-06-09.

[19] William Findlay, David Barrera, and Anil Somayaji. 2021. BPFContain: Fixing the Soft Underbelly of Container Security. *arXiv preprint arXiv:2102.06972* (2021).

[20] William Findlay, Anil Somayaji, and David Barrera. 2020. Bpfbox: Simple Precise Process Confinement with EBPF. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop* (Virtual Event, USA) *(CCSW'20)*. Association for Computing Machinery, New York, NY, USA, 91–103. https://doi.org/10.1145/3411495.3421358

[21] Phani Kishore Gadepalli, Sean McBride, Gregor Peach, Ludmilla 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

[22] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1069–1084.

[23] WebAssembly Community Group. 2023. WebAssembly System Interface. https://github.com/WebAssembly/WASI accessed 2023-05-08.

[24] Andreas Haas, Andreas Rossberg, Derek L. Schuff, Ben L. Titzer, Michael Holman, Dan Gohman, Luke Wagner, Alon Zakai, and J. F. Bastien. 2017. Bringing the web up to speed with WebAssembly. In *PLDI*.

[25] Yutaro Hayakawa. 2018. eBPF implementation for FreeBSD. In *Proc. BSDCan*. 1–33.

[26] hBPF project. 2023. hBPF = eBPF in hardware. https://github.com/rprinz08/hBPF accessed 2023-06-09.

[27] Aaron Hilbig, Daniel Lehmann, and Michael Pradel. 2021. An Empirical Study of Real-World WebAssembly Binaries: Security, Languages, Use Cases. In *WWW*.

[28] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th international conference on emerging networking experiments and technologies*. 54–66.

[29] Wenjun Huang and Marcus Paradies. 2021. An Evaluation of WebAssembly and eBPF as Offloading Mechanisms in the Context of Computational Storage. *CoRR* abs/2111.01947 (2021). https://arxiv.org/abs/2111.01947

[30] IETF. [n. d.]. Charter for proposed Working Group BPF. https://datatracker.ietf.org/group/bpf/about/ accessed 2023-06-09.

[31] Abhinav Jangda, Bobby Powers, Emery D. Berger, and Arjun Guha. 2019. Not So Fast: Analyzing the Performance of WebAssembly vs. Native Code. In *USENIX ATC*.

[32] Jinghao Jia, YiFei Zhu, Dan Williams, Andrea Arcangeli, Claudio Canella, Hubertus Franke, Tobin Feldman-Fitzthum, Dimitrios Skarlatos, Daniel Gruss, and Tianyin Xu. 2023. Programmable System Call Security with eBPF. *arXiv preprint arXiv:2302.10366* (2023).

[33] 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 *IEEE Symposium on Security and Privacy (SP)*.

[34] The kernel development community. 2023. How to interact with BPF subsystem. https://docs.kernel.org/bpf/bpf_devel_QA.html accessed 2023-06-06.

[35] Ofek Kirzner and Adam Morrison. 2021. An Analysis of Speculative Type Confusion Vulnerabilities in the Wild. In *USENIX Security*.

[36] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, et al. 2020. Spectre attacks: Exploiting speculative execution. *Commun. ACM* 63, 7 (2020), 93–101.

[37] Daniel Lehmann, Johannes Kinder, and Michael Pradel. 2020. Everything Old is New Again: Binary Security of WebAssembly. In *USENIX Security*.

[38] Hongyi Lu, Shuai Wang, Yechang Wu, Wanning He, and Fengwei Zhang. 2023. MOAT: Towards Safe BPF Kernel Extension. *CoRR* abs/2301.13421 (2023).

[39] Soares Luis. 2023. eBPF: The Emerging Linux Kernel Technology Explained. https://medium.com/@luishrsoares/ebpf-the-emerging-linux-kernel-technology-explained-d9e86a3bf0ef accessed 2023-06-05.

[40] Andy Lutomirski. 2019. [WIP 0/4] bpf: A bit of progress toward unprivileged use. https://lwn.net/ml/linux-kernel/cover.1565040372.git.luto@kernel.org/ accessed 2023-06-10.

[41] Brian McFadden, Tyler Lukasiewicz, Jeff Dileo, and Justin Engler. 2018. Security chasms of wasm. *NCC Group Whitepaper* (2018).

[42] Microsoft. 2023. eBPF for Windows. https://microsoft.github.io/ebpf-for-windows/ accessed 2023-06-09.

[43] Shravan Narayan, Craig Disselkoen, Tal Garfinkel, Nathan Froyd, Eric Rahm, Sorin Lerner, Hovav Shacham, and Deian Stefan. 2020. Retrofitting Fine Grain Isolation in the Firefox Renderer. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 699–716. https://www.usenix.org/conference/usenixsecurity20/presentation/narayan

[44] Matherson Nate. 2021. What is eBPF? https://www.airplane.dev/blog/ebpf

[45] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. 2020. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*.

[46] Big Switch Networks. 2023. uBPF. https://github.com/iovisor/ubpf accessed 2023-06-09.

[47] Linux Kernel Organization. 2023. bpf-helpers.7 « man7 - man-pages/man-pages.git. https://git.kernel.org/pub/scm/docs/man-pages/man-pages.git/tree/man7/bpf-helpers.7 accessed 2023-06-09.

[48] Linux Kernel Organization. 2023. BPF maps — The Linux Kernel documentation. https://docs.kernel.org/bpf/maps.html accessed 2023-06-09.

[49] Linux Kernel Organization. 2023. eBPF Instruction Set Specification, v1.0. https://docs.kernel.org/bpf/instruction-set.html accessed 2023-06-09.

[50] Linux Kernel Organization. 2023. https://docs.kernel.org/bpf/btf.html. https://docs.kernel.org/bpf/btf.html accessed 2023-06-09.

[51] Linux Kernel Organization. 2023. verifier.c « bpf « kernel - kernel/git/torvalds/linux.git. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/kernel/bpf/verifier.c accessed 2023-05-24.

[52] Gragor Peach, Runyu Pan, Zhuoyi Wu, Gabriel Parmer, Christopher Haster, and Ludmila Cherkasova. 2020. eWASM: Practical Software Fault Isolation for Reliable Embedded Devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 39, 11 (2020), 3492–3505. https://doi.org/10.1109/TCAD.2020.3012647

[53] Samsung. 2020. Hardening Hostile Code in eBPF - Analysis on Kernel Self-Protection: Understanding Security and Performance Implication. https://samsung.github.io/kspp-study/bpf.html accessed 2023-05-12.

[54] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2022. Sound, precise, and fast abstract interpretation with tristate numbers. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. IEEE, 254–265.

[55] Perry Wagle, Crispin Cowan, et al. 2003. Stackguard: Simple stack smash protection for gcc. In *Proceedings of the GCC Developers Summit*. 243–255.

[56] Wasmer. 2023. Wasmer - Runtime. https://wasmer.io/products/runtime accessed 2023-06-09.

[57] Wasmtime. 2023. Config::consume_fuel in wasmtime - Rust. https://docs.wasmtime.dev/api/wasmtime/struct.Config.html#method.consume_fuel accessed 2023-06-09.

[58] Wasmtime. 2023. Security - Wasmtime. https://docs.wasmtime.dev/security.html accessed 2023-05-09.

[59] WebAssembly Community Group. 2023. Security - WebAssembly. https://webassembly.org/docs/security/ accessed 2023-05-09.

[60] WebAssembly Community Group. 2023. *WebAssembly Specification*. Draft Release 2.0 (Draft 2023-04-24). https://webassembly.github.io/spec/

[61] WebAssembly Community Group. 2023. WebAssembly W3C Process. https://github.com/WebAssembly/meetings/blob/main/process/phases.md original-date: 2017-05-04T04:32:02Z.

[62] Thomas Wirtgen, Tom Rousseaux, Quentin De Coninck, Nicolas Rybowski, Randy Bush, Laurent Vanbever, Axel Legay, and Olivier Bonaventure. 2023. xBGP: Faster Innovation in Routing Protocols. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. USENIX Association, Boston, MA, 575–592. https://www.usenix.org/conference/nsdi23/presentation/wirtgen