Unleashing Unprivileged eBPF Potential with Dynamic Sandboxing

Soo Yee Lim
University of British Columbia
Vancouver, British Columbia
Canada
sooyee@cs.ubc.ca

Xueyuan Han
Wake Forest University
Winston-Salem, North Carolina
USA
vanbasm@wfu.edu

Thomas Pasquier
University of British Columbia
Vancouver, British Columbia
Canada
tfjmp@cs.ubc.ca

ABSTRACT

For safety reasons, unprivileged users today have only limited ways to customize the kernel through the extended Berkeley Packet Filter (eBPF). This is unfortunate, especially since the eBPF framework itself has seen an increase in scope over the years. We propose SandBPF, a software-based kernel isolation technique that dynamically sandboxes eBPF programs to allow unprivileged users to safely extend the kernel, unleashing eBPF's full potential. Our early proof-of-concept shows that SandBPF can effectively prevent exploits missed by eBPF's native safety mechanism (i.e., static verification) while incurring 0%-10% overhead on web server benchmarks.

Note: This is a preprint version of the paper accepted at the 1st SIGCOMM Workshop on eBPF and Kernel Extensions [28].

KEYWORDS

eBPF, Dynamic Sandbox, Software Fault Isolation, Binary Rewriting

1 INTRODUCTION

The extended Berkeley Packet Filter (eBPF) enables users to extend the Linux kernel's capabilities without modifying its source code. To ensure safe extension of the kernel, eBPF uses a *verifier* to *statically* verify the safety of an eBPF program before it is executed in the kernel. Unfortunately, known vulnerabilities allow an eBPF program to circumvent static verification checks, which enables a malicious eBPF program to access arbitrary kernel memory [5, 7–9] or execute arbitrary kernel code [6].

Most Linux distributions [17, 18] err on the side of caution by allowing *only privileged users* to run eBPF programs. However, this restriction significantly limits the ability for non-privileged applications to customize the kernel. For example, it makes adopting emerging eBPF technologies to support the implementation of specialized audit [29], scheduling [26], and synchronization policies [37] in the kernel for a particular application or container difficult.

An alternative approach is to formally verify that the eBPF verifier *correctly* guarantees the absence of *all* possible attacks. The eBPF framework relies primarily on the verifier to ensure the safety of an eBPF program. The verifier inspects the program *at load time*, so it imposes no run-time performance overhead. However, prior incidents [4, 7–9, 11, 12] have repeatedly shown that a verified eBPF program is *not always safe*. The complexity of eBPF programs makes verifying these programs difficult, resulting in both *specification bugs* [4] (i.e., missing checks for a specific type of vulnerabilities) and *implementation bugs* [7–9, 11, 12] (i.e., incomplete checks for a supposedly checked vulnerability) in the verifier. As we elaborate in §2, formally verifying the eBPF verifier cannot simultaneously resolve both types of bugs.

In light of these challenges, we take a completely different approach to enabling unprivileged eBPF programs to safely run in the Linux kernel. We leverage software fault isolation (SFI) [41], a software-based kernel isolation technique, and binary rewriting to *dynamically sandbox* an eBPF program. Our approach, which we name SandBPF, prevents an eBPF program from committing a memory safety violation at run time by confining all memory accesses to within the eBPF sandbox and limiting eBPF control transfers to only valid call targets. The proof-of-concept implementation of SandBPF shows that dynamic sandboxing is effective in catching safety bugs that are missed by the verifier while incurring reasonable performance overhead in realistic settings: e.g., 0-10% on web server macrobenchmarks. These results are encouraging, especially since we took a pure software approach (as a first step to demonstrate efficacy), rather than leveraging hardware support for isolation [2, 3, 35] to improve performance, which we leave for future work.

Contributions.

- We study safety mechanisms in eBPF to motivate the need for *dynamic isolation* (§2).
- We demonstrate that dynamic sandboxing is a viable solution to address eBPF security concerns. Our changes are a self-contained extension to the kernel and require no modification to existing workflows or programs (§4).

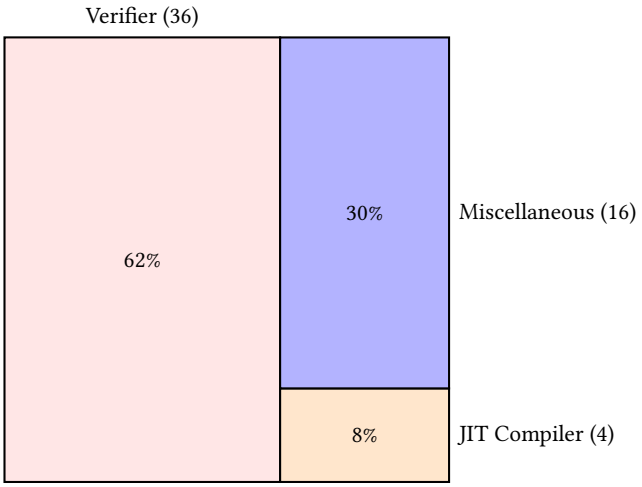

Figure 1: A tally of eBPF-related CVEs from 2010 to 2023. There are a total of 56 CVEs, the majority of which were discovered in the verifier.

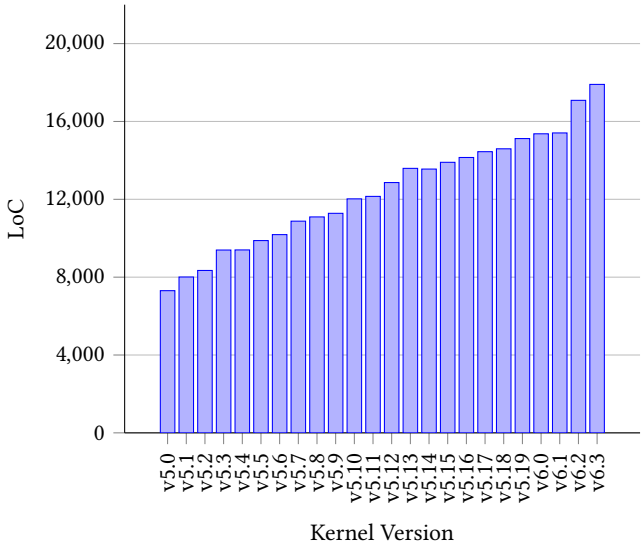

Figure 2: The evolution of the verifier's size in lines of code (LoC) from v5.0 in March 2019 (7,306 LoC) to v6.3 in April 2023 (17,904 LoC).

- We evaluate the performance overhead of our proof-of-concept implementation (§5).
- We discuss the limitations of our implementation and propose promising future research directions (§6).

Disclaimer: This work does not raise any ethical issues.

2 MOTIVATION

The eBPF verifier is the primary safety mechanism in eBPF, but it has also been a major source of vulnerabilities (Figure 1). Attackers can bypass the verifier and run malicious eBPF programs by exploiting specification or implementation bugs in the verifier. The verifier essentially works as a *blocklist* of prohibited behaviors; therefore, a specification bug exists when a specific type of exploitable vulnerabilities is not considered in the blocklist. For example, early versions of the verifier neglected alignment checks for stack pointers, which allowed adversaries to perform denial-of-service attacks [4]. Over the years, the verifier has grown significantly to mitigate specification bugs. Figure 2 shows that it has *more than doubled* in the last four years. Unfortunately, the ever-growing size and complexity makes it formidable to formally verify the verifier in its entirety to eliminate any implementation bugs. To the best of our knowledge, no existing work has proved the *completeness* of the verifier's specification or managed to formally verify its current (likely still incomplete) implementation. As a result, the Linux community has largely dismissed unprivileged eBPF programs as an unsafe feature that should not be used [16], despite their great potential.

We propose to rely on a *dynamic* enforcement mechanism, by rewriting the binary code to insert run-time checks, to *protect memory accesses* and *preserve control flow integrity*. Note that the verifier performs checks beyond memory accesses and control flow integrity (e.g., program termination). Thus, we still consider the verifier to be an important safety component of eBPF. However, our work can simplify the verifier's implementation (by removing memory access and control flow integrity checks), thereby reducing its codebase and easing its formal verification efforts. Our goal is to demonstrate the feasibility of dynamic enforcement; therefore, we leave the simplification of the verifier to future work.

3 THREAT MODEL

We assume that an adversary can run unprivileged eBPF programs. The adversary has no root access and thus is unable to load kernel modules or modify kernel code. However, they can exploit eBPF vulnerabilities to gain arbitrary read or write access to kernel memory, or execute arbitrary kernel code. We assume a W⊕X (write xor execute) enabled system, so the adversary cannot overwrite any executable pages.

Our trusted computing base includes the OS kernel (excluding the eBPF verifier and the JIT compiler) and `SandBPF`, whose correctness we plan to fully verify in future work. As shown in Figure 3, both the data of an eBPF program that resides in the sandbox provided by `SandBPF` and the original eBPF code are assumed to be untrusted. On the other hand, we assume `SandBPF`'s instrumentation and its own data stored outside of the sandbox (see details in §4) to be

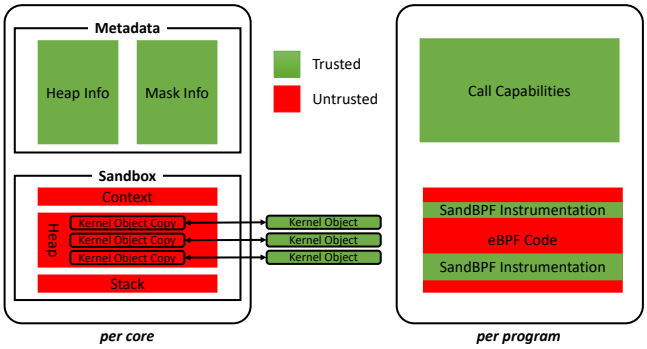

Figure 3: Illustration of the SandBPF design and its (un)trusted components.

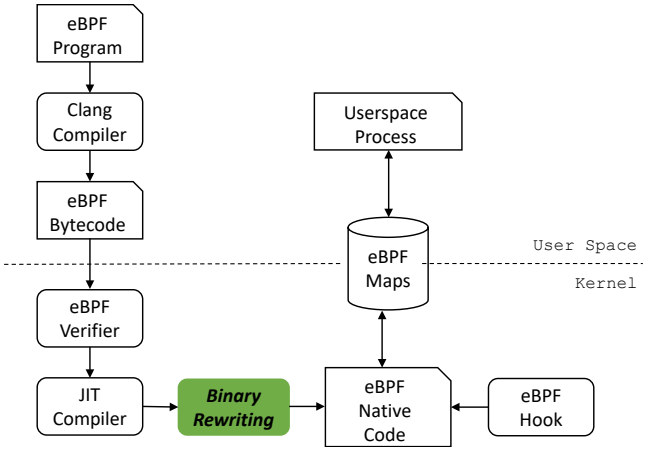

Figure 4: The workflow of a dynamically sandboxed eBPF program. SandBPF's binary rewriting is trusted.

trusted. We do not consider attacks originated from anywhere else in the kernel except eBPF. Note that SandBPF performs instrumentation on the final output of eBPF's JIT compilation; therefore, it does not rely on the correctness the JIT compiler or the verifier. Like in prior kernel isolation work [20–22, 31–34, 36, 38, 40], side-channel attacks are orthogonal and thus out of scope.

4 DESIGN & IMPLEMENTATION

Keeping our security mechanism completely transparent to eBPF programmers and ensuring compatibility with existing eBPF programs are the paramount design goals of SandBPF. As such, SandBPF is *minimally invasive*, reusing the existing eBPF pipeline and extending only what is necessary. Specifically, SandBPF adds binary rewriting only at the end of the JIT compilation, as shown in Figure 4. This allows experienced users to develop eBPF programs like they normally do, while new users can rely on existing eBPF documentation.

Our SandBPF proof-of-concept leverages software fault isolation (SFI) [41] to create a safe sandbox to execute eBPF programs. SFI is a software-based isolation technique that transforms memory-access and control-transfer instructions to prevent a program from accessing memory outside a designated region. Figure 3 shows our SFI design. SandBPF ensures memory safety and control-flow integrity via *address masking* and *trampolined control transfers*, respectively. The former restricts an eBPF program's access to only the memory within its address space, and the latter enforces that the program call only entry points on its allowlist (based on its capabilities) when jumping outside of its domain. These checks together confine an eBPF program to its own sandbox at run time.

In the rest of this section, we discuss SandBPF's design in detail. In §6, we discuss the limitations of our current implementation.

4.1 Memory Access Checks

SandBPF enforces memory safety by masking the target addresses of all read and write instructions, so that all memory accesses, including the out-of-bounds ones, always fall within the data region of an eBPF domain. To avoid managing multiple address masks for data residing in different parts of the kernel address space, we reserve one memory page in each processor core to store the data of an eBPF program, as shown in Figure 3. We disable interrupt and preemption during the execution of an eBPF program, so each core runs only one eBPF program at a time.

We emit an address masking check on *every* read and write instruction. An address masking check consists of a bitwise-and instruction to clear the upper bits of the destination address, and subsequently a bitwise-or instruction to set the destination address to the memory region of an eBPF sandbox. For example, consider a 2048-byte aligned sandbox memory allocated at address `0xDEADB800`, and two pre-computed address masks (stored in the sandbox metadata as show in Figure 3): `and_mask` (`0x7FF`) and `or_mask` (`0xDEADB800`). If an attacker attempts to perform an out-of-bounds memory access at `0xDEAF1234`, address masking would transform the target address to `0xDEADBA34`, which falls within the sandbox. Thus, address masking guarantees that all memory accesses remain contained in the sandbox.

4.2 Accessing Kernel Objects

An eBPF program needs to access a kernel object (1) when its input pointer (i.e., "context") references a kernel object on program invocation, or (2) when an eBPF helper function returns a pointer to a kernel data structure. In the first case, we mirror the content of the data structure in the context

Table 1: The number of checks inserted and executed by SandBPF in our example programs.

Program	Injected		Executed	
	Address Masking	Trampoline	Address Masking	Trampoline
XDP	2	1	2	1
Socket Filter	12	10	12	10
Katran	641	42	35-37	1-2

region of the sandbox for the duration of the program execution. In the second case, we dynamically allocate heap space in the sandbox to store a copy of the object. As a concrete example, consider the eBPF ring buffer. We modify the ring buffer's reserve/commit mechanism to protect its access. On `bpf_ringbuf_reserve`, we create a buffer on the sandbox heap corresponding to the reserved region. Mapping information between the heap buffer and the reserved region is stored in the sandbox metadata, which is not accessible by the eBPF program. On `bpf_ringbuf_commit`, the sandbox copy is synced with its corresponding kernel object. This mechanism can be used for all similar operations. We discuss some security-related issues in §6.

4.3 Control Flow Integrity (CFI)

We enforce CFI by redirecting all call instructions to a trampoline that checks the validity of the destination operand. By design, eBPF programs can interact with the kernel only through an allowlist of helper functions. As different eBPF program types have access to different sets of helper functions, we associate each eBPF program type with a set of capabilities corresponding to the helper functions it is allowed to call. In other words, the capabilities specify the valid entry points for control transfers in an eBPF program, thereby preventing the program from executing arbitrary code in the kernel. The capabilities are computed once for every program type at load time and stored in a hash table to provide an $O(1)$ search time. SandBPF dynamically checks if the eBPF program has the capability to call the target.

5 EVALUATION

We implemented SandBPF for Linux 5.18.7. All experiments were performed on a bare metal machine with 32GiB of RAM and an 8-core, 2.3GHz Intel Core i7 CPU. We disabled hyperthreading, turbo boost, and frequency scaling to reduce variance in performance benchmarking. We ran each experiment on two kernel configurations: (1) The *vanilla* configuration runs on the unmodified kernel, as our baseline. (2) The *sandbox* configuration runs on the same kernel but is instrumented with SandBPF on eBPF programs.

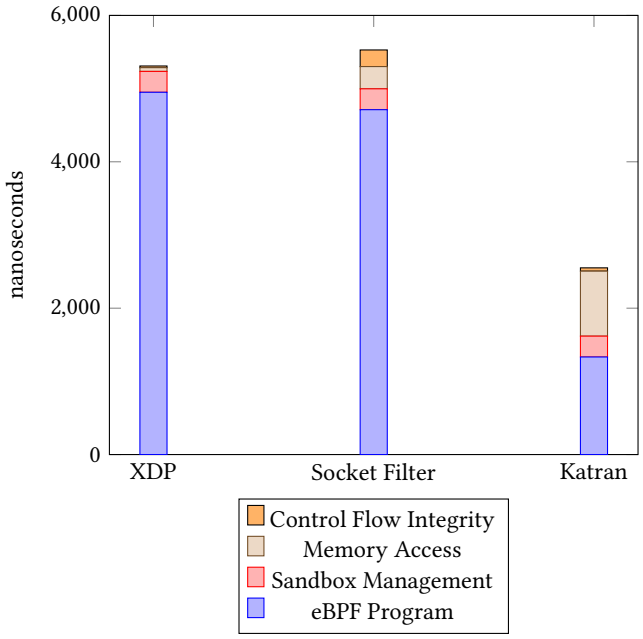

Figure 5: Breakdown of the overhead introduced by SandBPF on our three example eBPF programs.

5.1 Understanding Overhead

Table 1 shows the number of checks SandBPF inserted into sandboxed eBPF programs. Note that the number of *inserted* instrumentation points does not directly influence performance; rather, performance hinges on the number of checks actually *executed* at run time, which in turn depends on execution paths. In a complicated eBPF program, e.g., the Katran load balancer [24], we often observe fewer than 10% of the inserted checks being executed at run time.

In Figure 5, we run three eBPF programs: (1) The `XDP` program logs the size of each ingress packet entering a networking device. (2) The `Socket Filter` program monitors packets by attaching itself to the `sock_queue_rcv_skb()` function. It exchanges packet information with a userspace process through an eBPF ring buffer. (3) The `Katran` program performs load balancing and is attached to the NIC as an `xdp` program. `Socket Filter` and `XDP` are programs provided as part of `libbpf-bootstrap` [13], a set of publicly available eBPF example programs. We decompose the overhead introduced by SandBPF alongside three categories:

Sandbox Management: This corresponds to sandbox initialization and shutdown on the execution of an eBPF program (e.g., preparing metadata, copying program parameters, switching execution context in and out of the sandbox, etc.).

Memory Access: SandBPF instruments both read and write instructions to protect the confidentiality and integrity of

Table 2: Microbenchmark measuring the impact of SandBPF on network communications running for 360s. (sf: send file, c→s: client to server, s→c: server to client).

Test	XDP Program		Socket Filter Program	
	Vanilla	SandBPF	Vanilla	SandBPF
Unidirectional throughput (MB/s)				
TCP sf	40,588	35,363 (**13%**)	68,394	52,655 (**23%**)
TCP c→s	36,740	33,374 (**9%**)	62,112	45,392 (**27%**)
TCP s→c	36,626	33,383 (**9%**)	62,674	45,628 (**27%**)
UDP s→c	48,019	46,226 (**4%**)	81,214	60,850 (**25%**)
Round-trip transaction rate (transaction/s)				
TCP	102,169	86,416 (**15%**)	135,713	94,611 (**30%**)
UDP	118,409	101,826 (**14%**)	152,485	104,900 (**31%**)

kernel memory. This differs from most SFI systems [20, 21, 21, 31, 41] that instrument only writes for performance.

Control Flow Integrity: This overhead is due mostly to the cost of linked list traversal when SandBPF searches through a hash table of call capabilities. We plan to optimize SandBPF's CFI checking performance using a custom data structure.

We show the overhead in Figure 5. The overall overhead is a function of the number of checks executed in these programs:

$$C_{overall} = C_{mem}(N_{mem}) + C_{tram}(N_{tram}) + C_{manage}$$

where the overhead of memory access checks (C_{mem}) and that of CFI checks using the trampoline mechanism (C_{tram}) are functions of the numbers of executed checks, N_{mem} and N_{tram}, respectively. Both N_{mem} and N_{tram} correlate with the complexity of the programs themselves. On the other hand, the overhead of sandbox management (C_{manage}) is constant.

The XDP and `Socket Filter` programs perform tracing, while `Katran` performs computations to make load-balancing decisions. XDP and `Socket Filter` spend the majority of their time in helper functions, which are not instrumented, resulting in relatively low overhead. On the other hand, `Katran` spend most of its time performing computation within instrumented code, resulting in a higher number of memory access checks and therefore a proportionally larger overhead.

5.2 Microbenchmark

We use netperf [15] to measure the overhead imposed by SandBPF on network communications. We run XDP and `Socket Filter`[1] on both kernel configurations and measure the unidirectional throughputs and round-trip latencies for TCP and UDP. We see in Table 2 that the overhead ranges from 4% to 33%. We note that the high overhead in the microbenchmark results is largely the artifact of netperf stressing the network interface and therefore spending most of its execution time in kernel code that constantly triggers eBPF events. In practice, user applications typically perform meaningful computations in userspace, which would reduce the perceived SandBPF overhead as we show in §5.3.

[1]Due to space constraints, we leave an extensive evaluation plan involving other eBPF programs (which require more complex setups) to future work.

Table 3: Macrobenchmark measuring web server performance of 20-1000 concurrent connections.

Test	XDP Program		Socket Filter Program	
	Vanilla	SandBPF	Vanilla	SandBPF
Throughput (request/s)				
Apache 20	64,591	64,526 (**0%**)	62,269	60,089 (**4%**)
Apache 100	86,190	79,638 (**8%**)	87,576	83,751 (**4%**)
Apache 200	85,614	81,749 (**5%**)	85,671	83,381 (**3%**)
Apache 500	68,329	63,691 (**7%**)	72,399	67,177 (**7%**)
Apache 1000	66,472	62,508 (**6%**)	71,453	66,171 (**7%**)
Nginx 20	49,170	45,731 (**7%**)	50,095	45,331 (**10%**)
Nginx 100	58,613	54,494 (**7%**)	58,797	54,029 (**8%**)
Nginx 200	56,581	53,051 (**6%**)	58,447	53,869 (**8%**)
Nginx 500	50,495	47,699 (**6%**)	54,537	50,822 (**7%**)
Nginx 1000	46,302	44,977 (**3%**)	50,651	47,734 (**6%**)

5.3 Macrobenchmark

To evaluate the performance implication of SandBPF at the macro level, we select a set of macrobenchmarks (*Apache* and *Nginx*) from the Phoronix Test Suite [27] that characterize whole-system performance while stress-testing the network subsystems. These web server benchmarks measure network throughputs, which we report in Table 3. For all macrobenchmarks, we again run the XDP and `Socket Filter` programs to measure SandBPF's overhead on these workloads. We see that SandBPF incurs no more than 10% overhead.

5.4 Security

We tested our proof-of-concept implementation against two exploits that have publicly available source code (Table 4). These exploits leverage eBPF vulnerabilities to violate the confidentiality and integrity of kernel memory. Our experiments show that SandBPF can successfully prevent CVE-2021-3490 [9] and CVE-2021-4204 [10]. For example, in CVE-2021-3490, a bounds-tracking bug in the eBPF verifier leads to out-of-bounds access. An attacker can exploit this vulnerability to obtain arbitrary read and write accesses in the kernel memory. Consequently, they can leak `cred` pointers to userspace via eBPF maps and escalate privilege by overwriting the `cred` structure. To test both exploits, we ported SandBPF to the affected Linux kernel version v5.8.0-25.26. SandBPF successfully prevented both exploits through address masking. The attacker can no longer leak kernel pointers to perform subsequent malicious activity (i.e., privilege escalation).

Table 4: eBPF vulnerabilities that result in privilege escalation.

CVE	Vulnerability Description
CVE-2021-3490	The eBPF verifier incorrectly tracks the bounds of ALU32 bitwise operations, resulting in out-of-bounds reads and writes in the Linux kernel.
CVE-2021-4204	The eBPF verifier does not properly validate the bounds of `bpf_ringbuf_submit` and `bpf_ringbuf_discard` inputs, allowing out-of-bounds reads and writes in kernel memory.

6 DISCUSSION & FUTURE WORK

Performance. Our primary focus is to demonstrate the tremendous potential of using dynamic sandboxing to improve eBPF security and validate our approach through a proof-of-concept implementation. SandBPF used stock kernel functions (e.g., the standard memory allocator `kmem_alloc`) and data structures (e.g., hash maps), instead of any bespoke mechanisms to optimize its run-time performance. Moreover, we made no use of asynchronous mechanisms, nor did we restrict SandBPF's access checks to only `write` instructions. These "tricks" are often employed in prior work to reduce the cost of SFI [20, 21, 21, 31, 41]. Therefore, one could consider our current proof-of-concept implementation to be a *worst-case scenario* in terms of performance. Even so, we see only ≤10% overhead from SandBPF while providing fully-fledged memory and control flow protection on macrobenchmarks (§5.3). This overhead is a reasonable baseline for our future work to improve performance. For example, hardware features, such as ARM's Pointer Authentication Code (PAC) [1] and Memory Tagging Extension (MTE) [2], are promising avenues to expedite memory access checks (which constitute most of the overhead in non-tracing tasks as discussed in §5.1). We emphasize that at the moment, the *only* alternative to our approach is to entirely disable unprivileged eBPF programs.

Security. We backported SandBPF to earlier versions of the kernel to demonstrate its ability to safeguard vulnerabilities in the verifier (§5.4). While SandBPF's binary rewriting approach can reduce the kernel's attack surface, we recognize that our current evaluation is insufficient, given the innate complexity of proving the effectiveness of a sandboxing technique [19]. We are in the process of designing a more extensive evaluation methodology based on *fault injection*. We note that to fully unleash the potential of unprivileged eBPF programs, one must consider security issues beyond the ones addressed by SandBPF through dynamic sandboxing; vulnerabilities stemming from e.g., kernel namespacing and shared resources [23, 30, 42] will also need to be tackled. Furthermore, the verifier currently restricts an eBPF program's ability to modify certain attributes of some kernel objects (e.g., `sk_buff`). We plan to dynamically enforce such an restriction in the syncing mechanism described in §4.2.

Towards Simplifying the Verifier. Our approach makes it possible to replace a subset of the verifier's compilation-time checks, such as memory accesses (which are difficult and sometimes unsound to perform statically [7–9, 11, 12]), with SandBPF's dynamic checks. This not only simplifies the verifier by obviating the need to check aspects of correctness that have been proven hard to guarantee, but more importantly, allows the eBPF framework to relax some constraints imposed on its programs (e.g., in dynamic memory allocation). These constraints exist due to the difficulties of verifying statically the safety of an eBPF program. There is a wealth of opportunities to explore ways to relax eBPF constraints to enrich its functionality, which we leave to future work.

7 RELATED WORK

SandBPF leverages a software-based isolation technique, specifically SFI, to dynamically sandbox eBPF programs. SFI instruments code with dynamic checks to ensure that run-time data accesses and control flow transfers are within specified bounds. Prior work [20, 21, 31, 39] leveraged SFI to sandbox OS extensions such as device drivers. For example, BGI [20] associates an access control list with each byte of memory to specify byte-level access permissions. SandBPF instead enforces SFI at the page level, restricting the access of a sandboxed eBPF program to pre-defined pages of kernel memory. Due to performance concerns, most SFI systems [21, 31], including BGI, do not check read instructions; as a result, they cannot provide *confidentiality* guarantees. In contrast, to account for potential leakage of kernel memory to userspace, SandBPF instruments both read and write instructions, thus assuring both confidentiality and integrity of kernel memory. This design decision comes at the cost of performance as discussed in §6.

Other than SFI, prior work [43] also proposed to use implicit pointer bounds information to enforce fine-grained type and memory safety. For example, SafeDrive [43] allows developers to provide type annotations that describe pointer bounds to insert dynamic checks in device drivers. Unlike SafeDrive, SandBPF instrumentation is completely automated, requiring no manual annotation effort. This is possible thanks to the relatively well-defined eBPF API and a well-scoped set of kernel objects that an eBPF program

is usually allowed to interact with, as compared to Linux kernel modules or device drivers.

Recent work [25] has also proposed a new Rust-based eBPF design. It leverages the Rust tool-chain to perform static checks (e.g., memory safety and control-flow integrity). In addition, it uses run-time mechanisms to enforce properties such as program termination that Rust does not natively provide. While this approach eliminates the need for an in-kernel eBPF verifier, it has two main drawbacks. First, the Rust verification tool-chain must be executed by a trusted third party before signing the eBPF programs. As a result, this approach limits the kernel to load only eBPF programs that are signed by trusted third parties, as the kernel itself can no longer independently verify them. This runs contrary to SandBPF's philosophy of increasing eBPF usage as an end goal. Second, vulnerabilities in the complex Rust ecosystem [14] take us back to the same problem with the eBPF verifier – static analysis alone is insufficient to guarantee run-time safety of eBPF programs. In other words, eBPF extensions can exploit Rust verifier's vulnerabilities to corrupt kernel memory at run time. Therefore, we believe that dynamic sandboxing is a key step towards unleashing the potential of unprivileged eBPF.

8 CONCLUSION

We show that dynamic sandboxing is a viable approach to enforce a number of security properties in eBPF programs, complementary to the current static mechanism employed by the eBPF verifier. Dynamic sandboxing will not replace verification; instead, it enhances run-time safety of the kernel to justify the (currently dismissed) support of unprivileged eBPF programs. SandBPF, our proof-of-concept implementation based on software fault isolation, incurs reasonable overhead. We believe that our work opens up an interesting design space, which allows future work to bring competitive performance improvements to this approach, particularly by leveraging available hardware features.

ACKNOWLEDGMENT

We acknowledge the support of the Natural Sciences and Engineering Research Council of Canada (NSERC). Nous remercions le Conseil de recherches en sciences naturelles et en génie du Canada (CRSNG) de son soutien. This material is based upon work supported by the U.S. National Science Foundation under Grant CNS-2245442. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation.

REFERENCES

[1] [n. d.]. ARMv8.3 Pointer Authentication. ([n. d.]). https://lwn.net/Articles/718888/
[2] [n. d.]. ARMv8.5-A Memory Tagging Extension. Online (Accessed: August 2, 2023). ([n. d.]). https://developer.arm.com/documentation/102925/0100.
[3] [n. d.]. ARMv8.5-A Pointer Authentication. Online (Accessed: August 2, 2023). ([n. d.]). shorturl.at/GHM69.
[4] [n. d.]. CVE-2017-17856. Online (Accessed: August 2, 2023). ([n. d.]). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2017-17856.
[5] [n. d.]. CVE-2020-8835. Online (Accessed: August 2, 2023). ([n. d.]). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8835.
[6] [n. d.]. CVE-2021-29154. Online (Accessed: August 2, 2023). ([n. d.]). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-29154.
[7] [n. d.]. CVE-2021-31440. Online (Accessed: August 2, 2023). ([n. d.]). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-31440.
[8] [n. d.]. CVE-2021-33200. Online (Accessed: August 2, 2023). ([n. d.]). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-33200.
[9] [n. d.]. CVE-2021-3490. Online (Accessed: August 2, 2023). ([n. d.]). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3490.
[10] [n. d.]. CVE-2021-4204. Online (Accessed: August 2, 2023). ([n. d.]). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-4204.
[11] [n. d.]. CVE-2022-0264. Online (Accessed: August 2, 2023). ([n. d.]). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-0264.
[12] [n. d.]. CVE-2022-23222. Online (Accessed: August 2, 2023). ([n. d.]). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-23222.
[13] [n. d.]. ibbpf-bootstrap. ([n. d.]).
[14] [n. d.]. Mitre: Rust CVEs. Online (Accessed: August 2, 2023). ([n. d.]). https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=rust.
[15] [n. d.]. Netperf. Online (Accessed: August 2, 2023). ([n. d.]). https://hewlettpackard.github.io/netperf/.
[16] [n. d.]. Reconsidering unprivileged BPF. Online (Accessed: August 2, 2023). ([n. d.]). https://lwn.net/Articles/796328/.
[17] [n. d.]. Security Hardening: Use of eBPF by unprivileged users has been disabled by default. Online (Accessed: August 2, 2023). ([n. d.]). https://www.suse.com/support/kb/doc/?id=000020545.
[18] [n. d.]. Unprivileged eBPF disabled by default for Ubuntu 20.04 LTS, 18.04 LTS, 16.04 ESM. Online (Accessed: August 2, 2023). ([n. d.]). https://discourse.ubuntu.com/t/27047.
[19] Frédéric Besson, Sandrine Blazy, Alexandre Dang, Thomas Jensen, and Pierre Wilke. 2019. Compiling sandboxes: Formally verified software fault isolation. In *European Symposium on Programming (ESOP'19)*. Springer, 499–524.
[20] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. 2009. Fast byte-granularity software fault isolation. In *Symposium on Operating Systems Principles (SOSP'09)*. ACM, 45–58.
[21] Ulfar Erlingsson, Martín Abadi, Michael Vrable, Mihai Budiu, and George C Necula. 2006. XFI: Software guards for system address spaces. In *Symposium on Operating Systems Design and Implementation (OSDI'06)*. USENIX, 75–88.
[22] Vinod Ganapathy, Arini Balakrishnan, Michael M Swift, and Somesh Jha. 2007. Microdrivers: A new architecture for device drivers. *Network* 134 (2007), 27–8.
[23] Xing Gao, Zhongshu Gu, Zhengfa Li, Hani Jamjoom, and Cong Wang. 2019. Houdini's escape: Breaking the resource rein of linux control groups. In *Conference on Computer and Communications Security (CCS'19)*. ACM, 1073–1086.
[24] Meta Incubator. [n. d.]. Katran:A high performance layer 4 load balancer. Online (Accessed: August 2, 2023). ([n. d.]). https://github.com/facebookincubator/katran.

[25] Jinghao Jia, Raj Sahu, Adam Oswald, Dan Williams, Michael V Le, and Tianyin Xu. 2023. Kernel Extension Verification is Untenable. In *Workshop on Hot Topics in Operating Systems (HotOS'23)*. ACM, 150–157.

[26] Kostis Kaffes, Jack Tigar Humphries, David Mazières, and Christos Kozyrakis. 2021. Syrup: User-defined scheduling across the stack. In *Symposium on Operating Systems Principles (SOSP'21)*. ACM, 605–620.

[27] Michael Larabel and Matthew Tippett. 2011. Phoronix test suite. *Phoronix Media,[Online]. Available: http://www. phoronix-test-suite. com/.[Accessed June 2016]* (2011).

[28] Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. 2023. Unleashing Unprivileged eBPF Potential with Dynamic Sandboxing. In *SIGCOMM Workshop on eBPF and Kernel Extensions*. ACM.

[29] Soo Yee Lim, Bogdan Stelea, Xueyuan Han, and Thomas Pasquier. 2021. Secure Namespaced Kernel Audit for Containers. In *Symposium on Cloud Computing (SoCC'21)*. ACM, 518–532.

[30] Congyu Liu, Sishuai Gong, and Pedro Fonseca. 2023. KIT: Testing OS-Level Virtualization for Functional Interference Bugs. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'23)*. ACM, 427–441.

[31] Yandong Mao, Haogang Chen, Dong Zhou, Xi Wang, Nickolai Zeldovich, and M Frans Kaashoek. 2011. Software fault isolation with API integrity and multi-principal modules. In *Symposium on Operating Systems Principles (SOSP'11)*. ACM, 115–128.

[32] Derrick McKee, Yianni Giannaris, Carolina Ortega Perez, Howard Shrobe, Mathias Payer, Hamed Okhravi, and Nathan Burow. 2022. Preventing Kernel Hacks with HAKC. In *Network and Distributed System Security Symposium (NDSS'22)*. Internet Society.

[33] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, et al. 2019. LXDs: Towards Isolation of Kernel Subsystems. In *Annual Technical Conference (ATC'19)*. USENIX, 269–284.

[34] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2020. Lightweight kernel isolation with virtualization and VM functions. In *International Conference on Virtual Execution Environments*. ACM, 157–171.

[35] Gil Neiger, Amy Santoni, Felix Leung, Dion Rodgers, and Rich Uhlig. 2006. Intel virtualization technology: Hardware support for efficient processor virtualization. *Intel Technology Journal* 10, 3 (2006).

[36] Ruslan Nikolaev and Godmar Back. 2013. VirtuOS: An operating system with kernel virtualization. In *Symposium on Operating Systems Principles (SOSP'13)*. ACM, 116–132.

[37] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. 2022. Application-Informed Kernel Synchronization Primitives. In *Symposium on Operating Systems Design and Implementation (OSDI'22)*. USENIX, 667–682.

[38] Matthew J Renzelmann and Michael M Swift. 2009. Decaf: Moving Device Drivers to a Modern Language.. In *Annual Technical Conference (ATC'09)*. USENIX.

[39] Christopher Small and Margo Seltzer. 1998. MiSFIT: Constructing safe extensible systems. *IEEE concurrency* 6, 3 (1998), 34–41.

[40] Michael M Swift, Brian N Bershad, and Henry M Levy. 2003. Improving the reliability of commodity operating systems. In *Symposium on Operating Systems Principles (SOSP'03)*. ACM, 207–222.

[41] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. 1993. Efficient Software-based Fault Isolation. In *Symposium on Operating Systems Principles (SOSP'93)*. ACM, 203–216.

[42] Nanzi Yang, Wenbo Shen, Jinku Li, Yutian Yang, Kangjie Lu, Jietao Xiao, Tianyu Zhou, Chenggang Qin, Wang Yu, Jianfeng Ma, et al. 2021. Demons in the shared kernel: Abstract resource attacks against os-level virtualization. In *Conference on Computer and Communications Security (CCS'21)*. ACM, 764–778.

[43] Feng Zhou, Jeremy Condit, Zachary Anderson, Ilya Bagrak, Rob Ennals, Matthew Harren, George Necula, and Eric Brewer. 2006. SafeDrive: Safe and recoverable extensions using language-based techniques. In *Symposium on Operating Systems Design and Implementation (OSDI'06)*. USENIX, 45–60.