SafeBPF: Hardware-assisted Defense-in-depth for eBPF Kernel Extensions

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

Tanya Prasad
tanyapsd@cs.ubc.ca
University of British Columbia
Vancouver, British Columbia, Canada

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

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

ABSTRACT

The eBPF framework enables execution of user-provided code in the Linux kernel. In the last few years, a large ecosystem of cloud services has leveraged eBPF to enhance container security, system observability, and network management. Meanwhile, incessant discoveries of memory safety vulnerabilities have left the systems community with no choice but to disallow unprivileged eBPF programs, which unfortunately limits eBPF use to only privileged users. To improve run-time safety of the framework, we introduce SafeBPF, a general design that isolates eBPF programs from the rest of the kernel to prevent memory safety vulnerabilities from being exploited. We present a pure software implementation using a Software-based Fault Isolation (SFI) approach and a hardware-assisted implementation that leverages ARM's Memory Tagging Extension (MTE). We show that SafeBPF incurs up to 4% overhead on macrobenchmarks while achieving desired security properties.
Note: This is a preprint of the paper accepted at the 2024 ACM Cloud Computing Security Workshop [78].

1 INTRODUCTION

In recent years, we have been witnessing an increasing uptake, both in academia and industry, of using the extended Berkeley Packet Filter (eBPF) to customize in-kernel behavior. The eBPF framework is designed to enable safe extension of the Linux kernel without modifying the kernel source code. Prior work [49, 52, 79, 88, 90, 91, 100, 101] has demonstrated success of leveraging eBPF in a variety of use cases, ranging from packet forwarding to balance network traffic load [31] and network monitoring to secure and troubleshoot communications in a microservices architecture [15, 39], to application-specific scheduling [70], prefetching [52] and page cache management [73]. Companies like Tigera [44] and Cilium [15] have capitalized on eBPF to deliver container security, networking, and observability solutions for modern cloud computing environments. However, unprivileged containers are often unable to leverage these eBPF features as they require privileges not granted to untrusted containers.

Indeed, the safety of the eBPF framework relies primarily on *statically* verifying an eBPF program before it is allowed to run in the kernel. Unfortunately, recent work [68, 77] has shown that static verification alone is insufficient to prevent malicious eBPF programs from accessing arbitrary kernel memory [17–21, 23–25]. Consequently, most kernel distributions disable the use of eBPF by unprivileged users [42, 46]. This significantly limits its adoption and sometimes even encourages deliberate, unsafe practices, thereby defeating the purpose of disallowing unprivileged uses in the first place. For example, practitioners are often interested in running eBPF programs in unprivileged containers. With privileged-only execution restriction in place, the general consensus is to circumvent this restriction using privileged processes, rather than finding safer alternatives. While this is only an anecdotal example, there is no denying that security, when done at the expense of usability (or convenience), often loses its priority.

If we improved the eBPF verifier, then this could potentially guarantee the safety of unprivileged eBPF programs and therefore address the concerns that (rightfully) hinder eBPF's wide deployment. However, latest work [68] has shown that such efforts, like fuzzing [27, 30] and formal verification [50, 95, 96], are insufficient, due to the constantly increasing complexity of the verifier. A major overhaul of completely retiring the current verifier and using instead a memory-safe language like Rust [68] only shifts the problem from the verifier to the external Rust toolchain.

We introduce SafeBPF, a *dynamic sandboxing* approach that works alongside the verifier to improve eBPF security by isolating eBPF programs from the rest of the kernel. Using a combination of *software-based fault isolation* and *hardware-implemented memory tagging* techniques, SafeBPF confines all memory accesses of an eBPF program to a well-defined sandbox, thus preventing run-time violations of spatial memory safety, even if the vulnerabilities that lead to these violations bypass static verifier checks. Our evaluation shows that SafeBPF can effectively prevent memory bugs missed by the eBPF verifier while incurring at most 4% performance overhead on marcobenchmarks.

Contributions

- We propose a new execution environment for eBPF extensions (§4) and explore different mechanisms to dynamically enforce spatial memory safety (§5).
- We systematically evaluate SafeBPF and show that it introduces low run-time performance overhead while improving eBPF security (§6 and §7).
- To the best of our knowledge, we are the first to directly compare software-based isolation and ARM's MTE as alternative mechanisms to achieve in-kernel isolation (§6.2.1).
- We make our SafeBPF implementation and the corresponding patches to the Linux kernel publicly available for the community

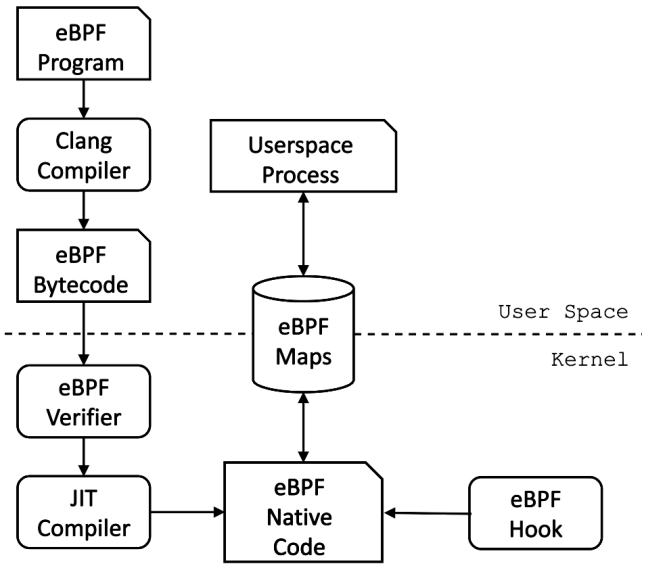

Figure 1: An overview of eBPF workflow.

to expand upon. We also make the materials to reproduce the evaluation available online.[1]

2 BACKGROUND & MOTIVATION

The extended Berkeley Packet Filter (eBPF) is a Linux framework that enables users to extend the kernel's capabilities without modifying its source code or loading additional kernel modules. The original BPF was designed for packet filtering; its functionality has since been extended as the underlying technology to drive a wide array of applications in areas such as performance monitoring [39], system tracing [6], load balancing [31], and security [26].

Kernel extensions, including those enabled by eBPF, can pose security risks to a system. Due to a lack of isolation in a monolithic operating system (OS), a vulnerability in a kernel extension grants an attacker full access to the rest of the OS with which the extension shares an address space. While eBPF is supposed to ensure safety through static verification, *run-time safety* remains an open problem (§2.3).

2.1 An Overview of eBPF

eBPF programs can be written in many high-level programming languages, such as C and Rust. Fig. 1 shows the general workflow of an eBPF program written in C. First, an eBPF program is compiled by Clang/LLVM to generate an ELF binary that contains architecture-independent eBPF bytecode. An eBPF ELF loader (e.g., libbpf) then parses the ELF binary and does the heavy lifting of preparing and loading the eBPF program into the Linux kernel. During loading, the kernel's eBPF verifier statically checks the safety of the eBPF bytecode. A Just-In-Time (JIT) compiler then translates the verified eBPF bytecode into native machine code, so the eBPF program can run as efficiently as natively compiled kernel code. Once the JIT compilation is completed, the eBPF program is marked as read-only to prevent any corruptions throughout its lifetime. Finally, the program is attached to its designated kernel hook point where it gets triggered and executed at run time. An eBPF program can interact with userspace processes via special shared data structures called eBPF maps. It is also restricted to a well-defined interface, i.e., the eBPF helper functions, to interact with the kernel. Depending on the program type, an eBPF program can access only a subset of the helper functions.

[1]Software artifacts (under GPLv2 license) and materials to reproduce the evaluation are available online at https://s00y33.github.io/publication/safebpf/SafeBPF.patch. We provide kernel patches for software-based isolation on both the x86 and ARM architecture, and hardware-assisted isolation on ARM.

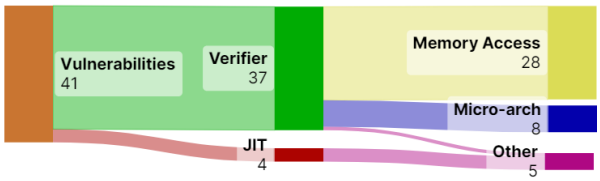

Figure 2: A summary of the types of CVEs reported in each component of eBPF from 2010 to 2023.

2.2 The eBPF Verifier

The eBPF verifier performs a two-pass static verification of the eBPF bytecode. The first pass conducts a depth-first search to reject eBPF programs that contain unreachable instructions, unbounded loops, or out-of-bounds jumps. It also rejects eBPF programs that are too large (exceeding 4,096 instructions for unprivileged programs and 1M for privileged ones) for the verifier to perform static analysis. Some of these restrictions are incompatible with compiler optimizations [14]; as such, correctly compiled eBPF programs can be rejected by the eBPF verifier. The proposal to integrate the eBPF verifier into the compiler could potentially address this issue [35], but maintaining such a compiler is difficult since the eBPF framework is constantly evolving at a fast pace (see §2.4).

The second pass simulates program execution, tracking its state (i.e., registers and stack) changes to catch unsafe operations (e.g., out-of-bounds accesses). On entry to each instruction, each register is assigned a type; the simulation of instruction execution changes the types of the registers depending on instruction semantics. For example, the eBPF verifier forbids an instruction from adding two pointers. Doing so would result in a register state of type `SCALAR_VALUE`, indicating a non-pointer register value. Any subsequent instruction attempting to access the register value as a pointer would then be rejected by the verifier.

2.3 eBPF Lacks Run-time Memory Safety

Vulnerabilities in the eBPF verifier often lead to bypasses of static security checks by the verifier, many of which have already been recorded in the Common Vulnerabilities and Exposures (CVE) database. Those bugs are related to *speculative execution* and *memory safety*. While general kernel hardening [45] and specific eBPF changes [3] have, for the most part, addressed the former concern, the latter, which comprises the majority of discovered vulnerabilities (28 out of 41, approximately 68%, see Fig. 2), remains a critical issue. Memory safety can be enforced by *dynamic* security checks;

unfortunately, these checks are absent, since verified eBPF programs run in the same address space as the rest of the kernel and are assumed to be trusted and safe. Adversaries exploiting memory-unsafe eBPF programs could therefore gain full access to kernel memory at run time. For example, CVE-2021-3490 is an out-of-bounds access vulnerability due to a bounds tracking bug in the eBPF verifier's 32-bit arithmetic and logic unit. This vulnerability grants unprivileged adversaries arbitrary read and write access to kernel memory, which enables them to achieve privilege escalation by overwriting the `cred` structure.

As eBPF continues to grow in complexity, with new features such as eBPF tokens [12] and eBPF exceptions [9] being regularly incorporated into the mainline Linux kernel, we can expect only *more* vulnerabilities to be reported in the future. More concerningly, a recent study [48] shows that vulnerabilities remain in the kernel for an average of 1,800 days before being addressed. The volume and duration of vulnerabilities inevitably puts *run-time* safety of eBPF extensions at a high risk. Not surprisingly, unprivileged eBPF is by default turned off on most Linux distributions [42, 46].

2.4 Improving the eBPF Verifier Is No Panacea

Many eBPF hardening solutions focus on improving the eBPF verifier. One approach is to formally verify the soundness of the verifier [50, 95, 96]. While this can eliminate implementation bugs, existing solutions verify only parts of the verifier, such as the tristate numbers (tnums) abstract domain [95] and range analysis [50, 96]. It is in fact challenging to extend formal verification to the entire verifier, because 1) the verifier itself has no formal specification [62]; 2) its size has increased significantly to verify the safety of the growing set of eBPF features (§2.3), which complicates formal verification and makes it hard to scale; and 3) emerging compiler features work directly against the verifier [14]. As we can see in Fig. 3, the size of the eBPF verifier has *more than doubled* in the past four years, creating ample opportunities for attackers to find bugs to bypass static checks and weaponize eBPF programs.

Jia et al. [68] propose to replace the eBPF verifier with the Rust compiler. Rust is a memory-safe programming language that leverages the type system and an ownership model to eliminate memory safety bugs at compile time. However, the Rust ecosystem is large and complex (approximately 1.6M lines of Rust code). Vulnerabilities in Rust [36] lead us back to the same problem with the eBPF verifier: *static checks alone cannot ensure run-time memory safety*. Moreover, since the proposed Rust-based eBPF design completely retires the eBPF verifier, it delegates the role of authorizing safe eBPF programs to a trusted third party in userspace. As a result, the kernel can load only eBPF programs signed by those parties, as it can no longer independently verify them. This design limits eBPF usage, sacrificing kernel extensibility for security. We note also that in a similar space, the driver signature scheme has been exploited by attackers [22, 28].

2.5 Our Proposed Approach

To bridge the security gap at run time, we propose to dynamically isolate eBPF programs, in addition to static verification. Since the majority of the eBPF verifier's vulnerabilities stem from bounds checking, we explore both software- and hardware-based isolation techniques to prevent run-time out-of-bounds kernel memory access. In particular, SafeBPF confines all memory accesses to within the eBPF sandbox, where all eBPF data resides. We show that eBPF isolation is feasible with software-based fault isolation, and its performance overhead can be reduced when using ARM Memory Tagging Extension (MTE). We discuss the applicability of our approach to other CPU architectures in §7.

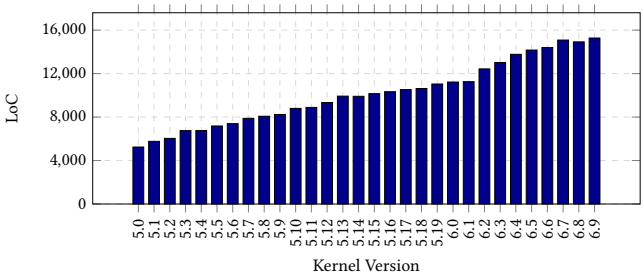

Figure 3: The evolution of the eBPF verifier's size in source lines of code (SLOC) from v5.0 in March 2019 (5,245 SLOC) to v6.9 in May 2024 (15,274 SLOC).

3 THREAT MODEL

SafeBPF targets *spatial memory safety*. In particular, we assume adversaries running *unprivileged* eBPF programs without root access, thus unable to load kernel modules or modify kernel code. However, they can exploit vulnerabilities in the eBPF verifier to bypass memory access checks, therefore gaining arbitrary read or write access to kernel memory. We assume a W⊕X (write xor execute) enabled system, so attackers cannot overwrite any executable pages.

Our trusted computing base includes the OS kernel (excluding the eBPF verifier) and SafeBPF. More specifically, as shown in Fig. 4, we assume that all the data in an eBPF program, including both private (e.g., stack and context) and shared data (e.g., eBPF maps), are untrusted. We also assume that SafeBPF's instrumentation and its own data, which are stored outside of the sandbox (see details in §4), are trusted.

SafeBPF instruments the final output of eBPF's JIT compilation; therefore, it does not rely on the correctness of the JIT compiler or the verifier. However, since we require eBPF programs to be *compiled*, we disable the legacy eBPF interpretation feature [10] to enforce JIT compilation. Note that eBPF interpretation is insecure due to its own set of vulnerabilities [69] and has already been disabled by default on the x86 and ARM architectures for most distributions. Securing eBPF interpretation is beyond the scope of this paper.

Like in prior kernel isolation work [53, 59, 60, 81, 82, 85–87, 89, 94], side-channel attacks are orthogonal and thus out of scope (see further discussion in §7).

4 SANDBOX DESIGN

Isolating an eBPF program involves 1) creating a sandbox and 2) enforcing isolation. In this section, we describe at a high level how SafeBPF constructs a sandbox for an eBPF program (Fig. 4). We then

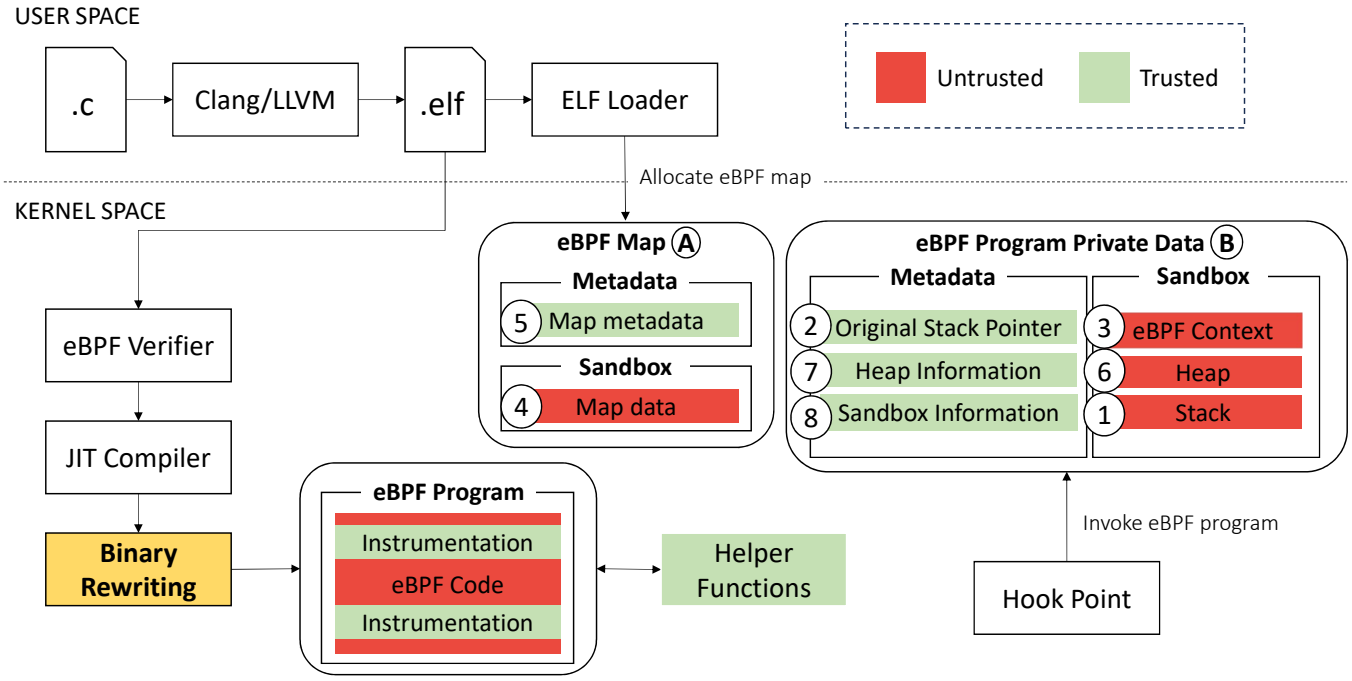

Figure 4: An illustration of the SafeBPF design.

detail different mechanisms to enforce isolation in §5 and compare their performance in §6.

We emphasize that SafeBPF is *not* intended to replace the eBPF verifier; in fact, we still rely on it to perform static checks (e.g., type checking the eBPF helper functions) and post-verification rewrites. However, in addition to these checks, SafeBPF fortifies *defense-in-depth* by dynamically isolating eBPF programs to ensure spatial memory safety at run time, thereby protecting the kernel from most known eBPF vulnerabilities.

4.1 Requirements

We design SafeBPF with the following first-order requirements in mind. The remainder of §4 and §5 explain how we successfully achieve these requirements.

Isolation. SafeBPF must isolate the kernel from eBPF exploits. It must prevent run-time violations of spatial memory safety and stop malicious eBPF programs from corrupting or leaking arbitrary kernel memory to userspace.

Efficiency. Performance is crucial to eBPF, especially since certain eBPF program types are usually attached to critical code paths in the kernel. For example, an XDP program is attached to a network interface card to process network packets. SafeBPF only minimally affects the overall performance of the kernel, incurring at most 4% overhead on the Apache benchmark. We justify this reasonably small performance cost by the substantial security *and* usability benefit to the eBPF framework, particularly since unprivileged eBPF programs are currently disallowed in most Linux distributions due to the risks associated with spatial memory safety.

Portability. SafeBPF can run on any x86-64 and ARM64 platforms. While SafeBPF leverages hardware extensions, its design also supports a fully software-based approach when the required hardware feature is unavailable. We carefully compare the overhead introduced by our software and hardware techniques in §6.

Minimally Invasive. SafeBPF is seamlessly integrated into the current eBPF pipeline, extending only what is necessary. Experienced users can thus develop eBPF programs like they normally would, and new users can still rely on the existing documentation. SafeBPF supports existing eBPF programs *as-is* and is entirely transparent to the end-user.

4.2 Sandbox Construction

SafeBPF identifies its sandbox's isolation boundary without users' manual annotation. Precisely defining the isolation boundary is important, since it determines what kernel data is and is not accessible to an eBPF program and therefore has security implications. Meanwhile, it is also a challenging problem, often addressed manually [53, 81, 82, 85, 86, 94, 99, 102] or semi-automatically [61, 65, 89] in prior dynamic sandboxing work. However, this is not necessary in SafeBPF thanks to the eBPF framework's already well-defined API and rules around the kinds of data that different types of eBPF programs are allowed to access. We use this information to define our isolation boundaries.

More specifically, since eBPF programs reside in the same virtual address space as the kernel, SafeBPF creates a *logically separated* sandbox in the kernel address space (Fig. 4). SafeBPF further divides the sandbox into different components based on data types: For a single eBPF program instance, one component stores the program's private data, i.e., its stack, heap, and context; every eBPF map also

has a separate component. Although eBPF programs are allowed to access other kernel objects by obtaining their pointers from helper functions (e.g., the `bpf_sk_lookup_tcp` helper function returns a TCP socket pointer to the eBPF program), SafeBPF does not protect accesses to these data as these features are not designed to be used by unprivileged users. In fact, so-called "offensive" eBPF features (e.g., the helper function `bpf_probe_read_user` that allows eBPF programs to read the memory of any process) should not be considered for container use because they can be exploited to break resource isolation in OS-level virtualization [63].

Each component is associated with a *metadata* region that is inaccessible to eBPF programs; metadata are used by the kernel (and SafeBPF) to manage the associated data objects. We compartmentalize the sandbox this way for two reasons. First, eBPF maps have a different lifetime than that of program-private data. While an eBPF map is freed only when no reference to it exists, data of the other types will immediately go out of scope upon a program's exit. By isolating them in separate components, SafeBPF can manage the life-cycle of a sandbox's components based on the lifetime of the data they isolate. Second, since eBPF maps can be shared among multiple eBPF programs while the other data types cannot, separating them makes it easy for SafeBPF to allow shared access to maps while restricting the other components to only their corresponding eBPF program instances. We detail how SafeBPF constructs the sandbox next.

4.2.1 eBPF Maps. eBPF maps (Ⓐ in Fig. 4) are memory regions shared among multiple eBPF programs and between an eBPF program and a userspace application. They must be accessed through dedicated helper functions. The eBPF framework implements various types of maps with different semantics (e.g., hash, array, and bloom filter). Along with map data, the framework allocates metadata (⑤ in Fig. 4), such as the reference counter and synchronization primitives, to manage a map and its specific semantic. Regardless of the type of an eBPF map, SafeBPF applies the same isolation principle. Specifically, when an eBPF map is allocated, which takes place before an eBPF program is loaded into the kernel, SafeBPF adds additional metadata (in the case of software-based isolation, see §5.2) or tags the map's memory region (in the case of hardware-assisted isolation, see §5.3) to ensure that the eBPF program is allowed to access only data in the map (④ in Fig. 4).

4.2.2 eBPF Program Private Data. Three types of private data could exist in an eBPF program:

eBPF Stack. SafeBPF places the entire eBPF stack in a sandbox so that an eBPF program operates on an isolated stack at run time (① in Fig. 4).

eBPF Context. The context, similar to function parameters, refers to the input passed to an eBPF program when it is invoked at a hook point. The eBPF framework provides developers with abstract data structures representing the underlying kernel objects that different types of eBPF programs can access. These data structures might also contain fields from objects that are pointed to by the kernel objects. On the other hand, not all fields of the represented kernel objects are included in the abstract data structures. For example, the context `__sk_buff` contains a subset of the fields in the kernel object `sk_buff`. One of its fields, `__sk_buff->ifindex`, corresponds to `sk_buff->dev->ifindex`. During the verification and compilation stage of an eBPF program, accesses to these abstract data structures are translated into direct accesses to the corresponding kernel objects. Unfortunately, this creates opportunities for an eBPF program to access fields of a kernel object that are not intended to be accessed from its abstract data structure. To address this issue, SafeBPF copies only the fields specified in the context to the sandbox so that the eBPF program can access only copied fields (③ in Fig. 4).

Dynamically-allocated Data. eBPF does not natively support dynamic memory allocation, because its verifier cannot resolve at compile time the bounds of dynamically-allocated memory. With dynamic sandboxing, SafeBPF enables dynamic memory allocation while ensuring its spatial memory safety at run time. The heap (⑥ in Fig. 4) resides in the sandbox, whereas its bookkeeping is isolated from eBPF programs in the metadata (⑦ in Fig. 4).

SafeBPF allocates one memory page per sandbox (Ⓑ in Fig. 4), since the current eBPF specification limits the stack size to 512 bytes. SafeBPF's sandbox size can be increased without any changes to its design if eBPF's required stack space increases. The first half of the page is dedicated to the metadata, and the second half is reserved for sandboxing an eBPF program's private data. The context is placed at the top of the sandbox, whereas the stack is placed at the bottom since it grows "downwards" from higher to lower addresses. The remaining space in the sandbox is used for dynamic memory allocation.

Upon the invocation of an eBPF program, SafeBPF prepares its context by copying to the sandbox only the fields of the context object that will be accessed by the program. In addition to the security advantage, this approach also improves run-time performance, as discussed in §6.2. For a context object with a nested structure (e.g., `__sk_buff` contains a pointer to the `bpf_sock` object), if the nested objects will be accessed by the program, SafeBPF recursively copies their fields (only those accessed by the program) to the dynamically-allocated memory space in the sandbox. As such, *all* pointers in the context point to addresses within the sandbox's heap. Upon the program's exit, SafeBPF updates the actual kernel object if any of its copied fields (including any nested objects) are modified in the sandbox.

4.3 Helper functions

eBPF programs interact with the system (e.g., printing debugging messages or using eBPF maps) through helper functions. Depending on the type of an eBPF program and the privileges held by the user loading it, helper functions accessible to the program vary. For example, users with the `CAP_PERFMON` privilege have access to performance monitoring helper functions.

Helper functions provide a secure, kernel-controlled mechanism to read or write kernel data structures. To support their use in the same way as the native eBPF framework, SafeBPF keeps the references to the original kernel objects in the metadata when it prepares the context upon program invocation (§4.2.2). This is necessary, because sandboxed eBPF programs operate on the copy of kernel object fields in the sandbox, whereas helper functions operate directly on kernel objects. When an eBPF program calls a helper function, SafeBPF transparently replaces the pointer to the sandboxed object

Composition	Lines of Source Code
Generic sandboxing code	1,361
SFI-specific code	348
MTE-specific code	80

Table 1: The size of SafeBPF codebase.

by the pointer to the original kernel object and syncs the data in their corresponding fields if needed. This also enables SafeBPF to mitigate *type confusion vulnerabilities* (e.g., CVE-2021-34866 [20]), where the pointer passed to a helper function is not of the type expected by the function. This type of vulnerabilities is common in languages like C with weak memory safety guarantees.

5 SANDBOX ENFORCEMENT MECHANISMS

We describe two sandboxing approaches, a software-only one based on address masking (§5.2) and a hardware-assisted one based on memory tagging (§5.3), to enforce isolation described in §4. We also propose an alternative hardware-assisted approach with lower performance overhead, albeit weaker security guarantees (§5.4). As shown in Table 1, 76% of SafeBPF's code is independent of the enforcement mechanism, while only 24% depends on the specific mechanism.

5.1 Sandbox Management

Recall in §4.2 that SafeBPF's sandbox protects two types of memory regions, eBPF maps and program private data. Each memory region is managed by a separate component in the sandbox, and each component contains the sandboxed data accessible by eBPF programs and the metadata required to manage the component. In this section, we detail the implementation of the metadata for each component. The next three sections discuss how SafeBPF's isolation mechanisms ensure only sandboxed data is accessible to eBPF programs.

eBPF Maps. The metadata are identical in both SafeBPF and the original eBPF framework, except that SafeBPF adds additional masks when running software-based isolation (see §5.2). SafeBPF modifies the eBPF maps' allocation logic to ensure the alignment of the sandboxed data satisfies the alignment requirements of the isolation mechanisms.

eBPF Program Private Data. The metadata contain the original stack pointer, the heap management data, and the sandbox management information. The sandbox management information describes the synchronized context objects (see §4.2.2) and, in the case of software-based isolation, the masks used to enforce isolation. Similar to eBPF maps, SafeBPF ensures the alignment requirements are met. To optimize performance, we reuse existing sandboxes to avoid constantly allocating new ones. We zero-out sandbox data before reusing a sandbox to prevent leakage of sensitive data.

SafeBPF instruments the prologue and epilogue of an eBPF program using binary rewriting to switch between the sandbox and the kernel. The prologue switches the stack pointer to point to the stack in the sandbox. It also saves the original stack pointer in the metadata to prevent the program from tampering with it (② in Fig. 4). Subsequently, the epilogue restores the original stack pointer to switch it back to the kernel stack upon program exit.

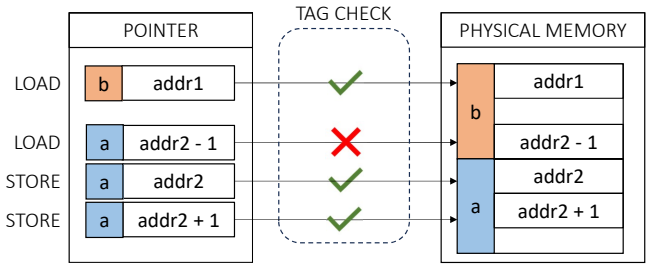

Figure 5: An overview of the MTE mechanism.

5.2 Software-based Isolation

SafeBPF uses *address masking*, a software fault isolation (SFI) based technique, to transform any memory address into an address in a memory region specified by a mask [71, 92]. By masking the *target* addresses of all load and store instructions in an eBPF program, SafeBPF enforces its spatial memory safety in such a way that *all* memory accesses of the program, including out-of-bounds accesses (if any), always fall within its sandbox.

Since SafeBPF's sandbox is separated into different components for two types of memory regions (§4.2), SafeBPF creates a pair of address masks, i.e., an `and_mask` and an `or_mask`, for each component during sandbox construction, and stores them in the component's metadata region. During JIT compilation, SafeBPF analyzes an eBPF program's information flow to differentiate between a memory access to a specific eBPF map and to a program instance's private data. It then inserts checks through binary rewriting using the corresponding address masks, which involves (1) a bitwise `and` operation between the `and_mask` and the target address to clear its upper bits, and (2) a bitwise `or` operation between the `or_mask` and the resulting address from (1) to map the address to within the sandbox. SafeBPF performs this binary rewriting in the last step of the compilation pipeline (Fig. 4). As an example, consider a 2048-byte aligned memory region of a sandbox component located at address `0xDEADB800`. SafeBPF computes its `and_mask` as `0x7FF` and `or_mask` as `0xDEADB800`. If an attacker attempted to access out-of-bounds memory at `0xDEAF1234`, address masking would transform the target address to `0xDEADBA34`, which would fall in the sandbox.

5.3 Hardware-assisted Isolation

ARM's Memory Tagging Extension (MTE) is a hardware primitive introduced in the ARMv8.5-A instruction set architecture. We illustrate how MTE works in Fig. 5. At a high level, MTE associates every 16 bytes of physical memory with 4-bit metadata known as a *memory tag*. It also modifies pointers to include in each pointer a *pointer tag* at bits 56-59 of the virtual address. The MTE-enabled CPU automatically and transparently checks if the pointer tag and the pointed memory's memory tag match on each load/store operation. A memory safety violation occurs when there is a mismatch between the two tags, and an exception is raised according to one of the three MTE's modes of operation: (1) *synchronous* - the kernel raises an exception synchronously upon a mismatch; (2) *asynchronous* - the kernel does not immediately raise an exception upon a mismatch but does so asynchronously; (3) *asymmetric* - loads are

handled in the synchronous mode and stores are handled in the asynchronous mode.

SafeBPF configures MTE to operate in the synchronous mode, so that spatial memory safety violations are immediately detected and their impact does not become observerable. SafeBPF turns on synchronous checks only during eBPF program execution and restores the original kernel settings upon entry/exit of the program and eBPF helper functions.

SafeBPF tags the sandbox memory regions with tag ⓐ during sandbox construction, while the rest of the kernel address space has a different tag ⓑ. Our tags are consistent with Linux's tagging convention [32], so that the kernel can access eBPF data, but an eBPF program cannot access kernel data outside of its sandbox.

Since MTE tags memory at a 16-byte granularity, we ensure sandbox memory regions are 16-byte aligned with their sizes rounded up to the nearest multiples of 16 bytes. SafeBPF also tags sandbox pointers with tag ⓐ so that subsequent sandbox accesses contain the same pointer tags as the memory tags. In contrast to address masking, this approach requires no instrumentation of load/store instructions, because the CPU transparently checks tags. If an attacker attempted to access memory outside of the sandbox, a tag mismatch would occur, which would raise an exception. SafeBPF would then initiate a kernel panic to stop the execution of the malicious eBPF program.

In this approach, SafeBPF still *copies* context objects that an eBPF program uses (§4.2.2). Alternatively, we could also leverage the hardware to avoid the cost by tagging these objects. We will explain its trade-offs next.

5.4 Alternative Hardware-assisted Isolation

We implement an alternative hardware-assisted approach that does not perform context synchronization but instead tags (and untags) objects (or a subsets of their attribute, when appropriate) accessed by an eBPF program upon entry/exit. Since we guarantee the execution of only one eBPF program *per core* (see §5.1), if the total number of tags is greater than the number of cores, we might be able to enforce exclusive access to tagged objects. However, if there are more cores than tags, we must reuse tags and inevitably weaken our security guarantees (see §7 for further discussion on the implications of limited tags in MTE). MTE's 16-byte tag granularity also becomes an issue in this alternative approach. Rather than tagging precisely the subset of an object's attributes that are accessed, we might have to tag beyond them (e.g., when an accessed attribute is a 4-byte integer), or sometimes even beyond an object's boundary (e.g., due to the alignment requirement). We could address this issue by modifying object layouts and alignments, but the resulting performance and memory implications would directly contradict our design goals (see §4.1). In §6.2, we show that this approach leads to only minor performance gain; therefore, it does not justify the loss of security guarantees achieved in §5.3.

6 EVALUATION

We implement SafeBPF for the Linux kernel v6.3.8. We perform all the evaluation on the Fedora Linux Asahi Remix 39 distribution, on a Mac Mini with an Apple M2 Pro CPU with 10 3.5GHz cores, 16GB of RAM, and a 512GB SSD. We run experiments on three kernel configurations: (1) the `vanilla` configuration runs on the unmodified kernel as our baseline; (2) the software-based sandbox configuration is denoted by `sfi`; and (3) the hardware-assisted sandbox configuration using ARM MTE is denoted by `mte` and `mte-min` (for the alternative implementation described in §5.4). In this evaluation, we answer the following three research questions:
Q1. How does MTE improve run-time performance of sandboxing compared to a pure software approach? (see §6.2.1)
Q2. How much overhead does SafeBPF introduce in realistic workloads? (see §6.2.2)
Q3. Does SafeBPF reduce the attack surface introduced by the use of unprivileged eBPF programs? (see §6.3)

6.1 MTE Instruction Analogs

MTE is introduced in ARMv8.5-A, but it is not supported by any open, widely-available systems at the time of this writing. For example, we see no availability of the feature on Apple M1 and M2 CPUs (see §E). Google Pixel 8 supports MTE, but it is a closed system. Finally, while Amazon's second-generation Neoverse instances can use MTE, they are not yet accessible to the general public. We therefore implemented and tested a SafeBPF prototype on QEMU (see §6.3). However, running a reliable performance evaluation on QEMU is hard, because it does not accurately represent the clock cycle count of the actual processor, even if we use a supposedly cycle-accurate timing model [57]. Hence, to more accurately measure performance overhead, we leverage *instruction analogs*, which are also used by prior MTE-related studies [75, 82], to simulate MTE instructions (e.g., `ldg` and `stg` for loading and storing tags in memory) and approximate their CPU cycles and memory footprints.

Memory tagging in SafeBPF is a one-off operation that takes place only when sandboxes are allocated, either at boot time or on map creation. Therefore, run-time performance overhead stems from tag checking operations by the CPU, which involve *tag loading* and *tag comparison*. Similar to prior work [75, 82], we do not simulate the latter, since it is performed by the hardware [5] and therefore should incur no measurable overhead. For tag loading, SafeBPF emits instruction analogs for the tag loading instruction (`ldg`) upon every eBPF memory access to simulate its cost [5]. Note that these tag loading instruction analogs *overestimate* the cost associated with MTE by manipulating the 49–53 bits of pointers to simulate address tags and replacing tag loads with regular loads from memory [75, 82], while the actual MTE implementation includes optimizations such as tag caching [5]. Thus, our evaluation is *conservative*, overestimating SafeBPF's overhead rather than underestimating it.

6.2 Performance

In §6.2.1, we run six eBPF programs with varying functionalities including network analysis, packet processing, performance tracing, and security, as a microbenchmark to measure the overhead introduced by SafeBPF in fine granularity. §6.2.2 discusses the results of macrobenchmark experiments when running realistic server workloads.

6.2.1 Microbenchmark. Based on prior work [51, 64, 69], we select the following programs:

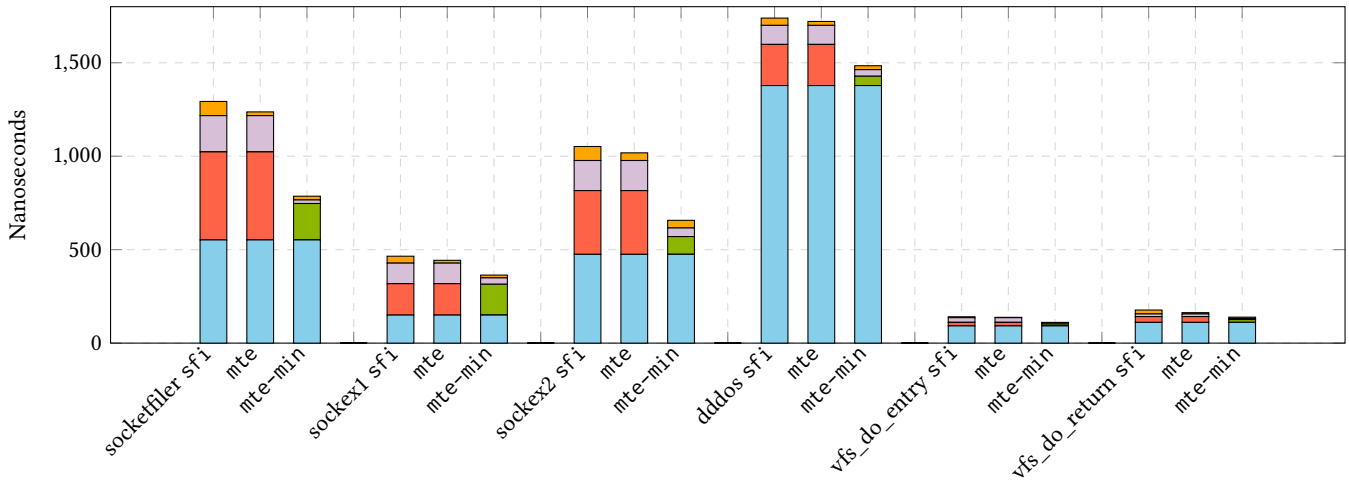

Figure 6: The overhead introduced by SafeBPF on six eBPF programs is divided into ■ *eBPF Program*, ■ *Context Synchronization*, ■ *Object Tagging*, ■ *Sandbox Management*, and ■ *Access Checks*.

`socketfiler` [33] attaches a socket BPF program to a hook called `sock_queue_rcv_skb()` to retrieve the protocol and the source and destination IP and port. This information is written to an eBPF ring buffer to be shared with a userspace program.
`sockex1` [34] counts the number of packets associated with a protocol observed on an attached network interface. The program retrieves the protocol attribute of a packet and increments the corresponding entry in an eBPF map.
`sockex2` [34] performs complex packet parsing. It uses IP addresses from `struct flow_keys->dst` (or hashes of IPv6 `dst`s) to tally the number of packets per IP address. The program can be attached to the Ethernet interface to print the statistics every second in userspace.
`dddos` [8] leverages `kprobe` to track packets arriving at an IP receive entry point (`ip_rcv`) and monitor the elapsed time between two received packets to detect potential denial-of-service attacks.
`vfs_do_entry` and **`vfs_do_return`** [8] attach to the entry and return of the `vfs_read` kernel routine, respectively. These two eBPF programs combined trace the latency of `vfs_read` and output it as a histogram distribution every five seconds. We refer to their combined overhead as `vfs_read_lat` in Table 2 and Table 3.

In Fig. 6, we compare the performance overhead between software-based (`sfi`), hardware-assisted isolation (`mte`), and the alternative hardware-assisted implementation described in §5.4 (`mte-min`). We make the following observations:

■ *Sandbox Management* overhead includes the cost of (1) switching between the original and sandboxed stack, (2) initializing the metadata of the heap in the sandbox, and (3) preparing the mapping between the original and the sandboxed context objects, which is needed for synchronization. This overhead is relatively small, depending neither on the number of load/store operations nor helper function calls.

■ *Access Checks* overhead is a function of the number of load/store operations in an eBPF program. Overall, the overhead is relatively small. Fig. 7 zooms in on it to help visualize the difference between

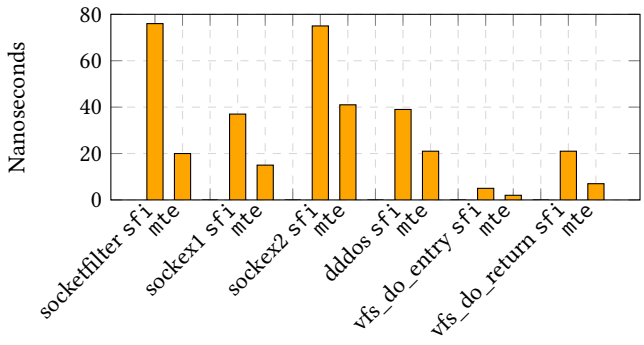

Figure 7: `sfi` vs `mte` ■ *Access Checks* cost. Note that `mte` and `mte-min` incur the same cost.

`sfi` and `mte`. Even with conservative MTE overhead measurement (in practice, the overhead would be lower), `mte` is 45%-73% faster than `sfi`.

■ *eBPF Program* measures the total execution time of the original program, which in most cases is dominated by the execution time of helper functions. SafeBPF adds no overhead *during* the execution of these functions (although a call to a helper function requires SafeBPF to handle context objects; this overhead is measured in ■ *Context Synchronization*, as discussed next). The nature of a helper function influences its execution time. For example, logging functions tend to have a longer execution time than those accessing data.

■ *Context Synchronization* dominates the overall overhead, which is consistent with the results from past work [85] on kernel compartmentalization. This overhead includes (1) allocating, at the beginning of execution, heap memory for context objects and those recursively pointed to by these objects; (2) handling calls to helper functions; and (3) copying synchronized objects' fields in and out of the sandbox during execution. The overhead depends on the

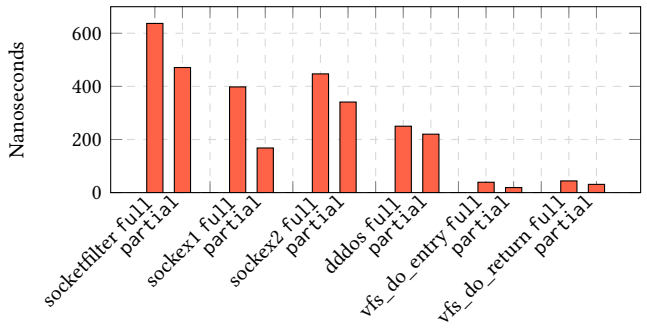

Figure 8: `full` vs `partial` ■ *Context Synchronization* cost.

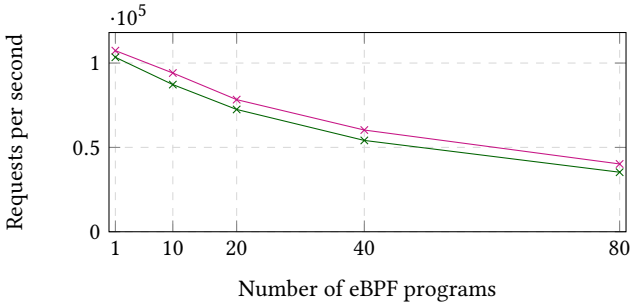

Figure 9: The overhead of SafeBPF as a function of the number of eBPF `sockex2` programs on the `Apache 200` workload. The standard deviations of all results are within 3%. We report performance results from the ■ *vanilla* kernel and from the SafeBPF kernel with ■ *mte* isolation.

complexity of context objects' data structures (e.g., whether they contain nested objects), the number of fields accessed in `read` or `write` mode, and the number helper function calls and their types. Fig. 6 reports the context synchronization cost for partial context, where we copy only fields accessed by eBPF programs. In Fig. 8, we compare the cost of synchronizing `full` and `partial` context. As discussed in §4.2.2, synchronizing partial context, in addition to improving security, also reduces overhead. Note that we could have avoided synchronization almost entirely in `mte` (§5.3), but doing so would degrade overall kernel performance and violate SafeBPF's design goal.

■ *Object Tagging* overhead is incurred only in the `mte-min` implementation (see §5.4) when it tags and untags kernel object(s) in and out of eBPF programs. This cost is a function of the size of the memory that needs to be tagged. Compared to `mte`, it reduces the overhead from two sources: (1) ■ *Sandbox Management* cost due to a more simplified sandbox setup, and (2) ■ *Context Synchronization* cost, as context synchronization is no longer needed.

Due to space constraints, we report only `mte` performance results in the remaining of the evaluation. The results of `sfi` and `mte-min` are available as supplementary material (§A and §B). However, we discuss all implementations in §6.2.3.

Netperf. We use Netperf [37] to measure SafeBPF's overhead on network communications. The benchmark measures unidirectional throughputs and round-trip latencies for TCP and UDP. Table 2 shows that SafeBPF introduces only 0%-7% overhead. This is significantly less than what one might extrapolate from Fig. 6, because the kernel spends only a fraction of the total time in an eBPF program when sending or receiving network packets.

6.2.2 Macrobenchmark. In Fig. 6, we show the overhead introduced by SafeBPF when executing eBPF programs *alone*. In absolute values, these overheads are less than 685*ns*. eBPF programs are generally triggered during the execution of system calls. Their execution time is largely dwarfed by the time spent in I/O, context switching, and user-space application logic. Consequently, we expect the overall degradation of a user-space application's performance to be minimal.

We run a set of macrobenchmarks from the Phoronix Test Suite [72] to measure the overhead introduced by SafeBPF on web server workloads (i.e., *Apache* and *Nginx*). The benchmark measures the number of requests/second processed with an increasing number of concurrent requests ($100 - 1,000$). As seen in Table 3, SafeBPF introduces 0-4% overhead.

In Fig. 9, we show the number of requests/second as a function of the number of loaded `sockex2` eBPF programs on the `Apache 200` workload from the Phoronix Test Suite. We select a single program-workload pair due to space constraints. The selected program-workload pair is the one where we observe the highest overhead in Table 3 to ensure that our performance reporting is conservative. SafeBPF performance degrades gracefully with 12% overhead when 80 programs are attached. In practice, it is unlikely that more than a couple of eBPF programs will be triggered on the same hook.

6.2.3 Comparison of Isolation Approaches. We see from Table 3, §A, and §B that the performance difference between the isolation strategies on macrobenchmark is minimal. As previously discussed, this is explained by the fact that the fraction of the time spent in executing eBPF programs is small compared to the overall execution time of typical userspace applications. MTE-based approaches fault on out-of-bounds accesses, unlike the SFI-based one, which silently restricts memory accesses within the sandbox. Faulting is the better behavior from a security perspective. Similarly, we see minimal practical performance improvement from the `mte-min` approach but weakened security guarantees. Finally, we emphasize that the overhead of both MTE-based approaches is overestimated due to the conservative nature of instruction analogs (see §6.1). Consequently, we believe SafeBPF's `mte` approach is practical.

6.3 Security

Preventing Known Vulnerabilities. We evaluate SafeBPF against seven *high severity* vulnerabilities, each of which has a publicly-available, working proof-of-concept exploit that leads to arbitrary kernel memory access and subsequently privilege escalation (Table 4). To test each exploit, we backport SafeBPF to a kernel version where the vulnerability is active; we make the corresponding kernel patches available for reproducibility. SafeBPF successfully prevents *all* exploits. For other memory access vulnerabilities in §2.3, we analytically confirm that they would be prevented by SafeBPF. Most of these vulnerabilities have no proof-of-concept exploits; developing them from high-level descriptions and Linux patches is

Test	sockfilter		sockex1		sockex2		dddos		vfs_read_lat	
	Vanilla	SafeBPF MTE	Vanilla	SafeBPF MTE	Vanilla	SafeBPF MTE	Vanilla	SafeBPF MTE	Vanilla	SafeBPF MTE
Unidirectional throughput (MB/s)										
TCP sf	95726	88950 (**7 %**)	119939	115170 (**4 %**)	114521	114486 (**0 %**)	100980	100741 (**0 %**)	126057	123267 (**2 %**)
TCP c → s	54641	53270 (**3 %**)	75081	75507 (**0 %**)	73581	72820 (**1 %**)	62237	61881 (**1 %**)	80307	80060 (**0 %**)
TCP s → c	54802	53063 (**3 %**)	81748	79079 (**3 %**)	75840	76147 (**0 %**)	62541	61718 (**1 %**)	80908	80445 (**1 %**)
UDP s → c	128181	121095 (**6 %**)	138650	139525 (**0 %**)	135468	135142 (**0 %**)	117237	118549 (**0 %**)	144860	140009 (**3 %**)
Round-trip transaction rate (transaction/s)										
TCP	29851	29455 (**1 %**)	30224	29552 (**2 %**)	30237	28973 (**4 %**)	27426	27032 (**1 %**)	28747	28761 (**0 %**)
UDP	32842	31429 (**4 %**)	33369	32750 (**2 %**)	33261	31838 (**4 %**)	29123	28679 (**2 %**)	31013	31379 (**0 %**)

Table 2: Netperf benchmark measuring the overhead of SafeBPF on network communications running for 360s. The standard deviations of all results are within 3% (sf: send file, c → s: client to server, s → c: server to client).

Test	sockfilter		sockex1		sockex2		dddos		vfs_read_lat	
	Vanilla	SafeBPF MTE	Vanilla	SafeBPF MTE	Vanilla	SafeBPF MTE	Vanilla	SafeBPF MTE	Vanilla	SafeBPF MTE
Requests per second (req/s)										
Apache 100	68808	67636 (**2 %**)	99391	95198 (**4 %**)	97782	94858 (**3 %**)	66868	66673 (**0 %**)	80745	78929 (**2 %**)
Apache 200	67862	67483 (**1 %**)	103741	100082 (**4 %**)	101212	96699 (**4 %**)	66770	65169 (**2 %**)	82804	79842 (**4 %**)
Apache 500	64510	63912 (**1 %**)	97828	97215 (**1 %**)	95597	92692 (**3 %**)	63170	62739 (**1 %**)	79111	78005 (**1 %**)
Apache 1000	63371	63496 (**0 %**)	95188	95292 (**0 %**)	94127	91388 (**3 %**)	64209	61715 (**4 %**)	78547	77203 (**2 %**)
Nginx 100	42376	41873 (**1 %**)	62514	61475 (**2 %**)	53609	52543 (**2 %**)	34615	33851 (**2 %**)	47831	46502 (**3 %**)
Nginx 200	42913	42569 (**1 %**)	62481	61612 (**1 %**)	53450	52482(**2 %**)	35050	34388 (**2 %**)	47759	46600 (**2 %**)
Nginx 500	42057	41813(**1 %**)	59563	58725 (**1 %**)	51174	50454 (**1 %**)	34895	34295 (**2 %**)	45896	44755 (**2 %**)
Nginx 1000	40804	40637 (**0 %**)	55778	55061 (**1 %**)	48063	47458 (**1 %**)	33470	33252 (**1 %**)	43224	42152 (**2 %**)

Table 3: Macrobenchmark measuring web server performance for 100-1000 concurrent connections.

CVE	Vulnerability Description	Mitigated
CVE-2023-2163 [25]	Incorrect verifier pruning causes the verifier to mark unsafe code paths as safe, which leads to arbitrary read/write in kernel memory, lateral privilege escalation, and container escape.	✓
CVE-2022-23222 [24]	The verifier incorrectly allows pointer arithmetic via certain `*_OR_NULL` pointer types, which allows out-of-bounds read/write in kernel memory. Local users can exploit this vulnerability to gain privilege.	✓
CVE-2021-4204 [23]	The verifier does not properly validate the bounds of inputs to `bpf_ringbuf_submit` and `bpf_ringbuf_discard`, thus allowing out-of-bounds read/write in kernel memory.	✓
CVE-2021-3490 [21]	The verifier incorrectly tracks the bounds of ALU32 bitwise operations that could lead to out-of-bounds read/write in kernel memory.	✓
CVE-2021-31440 [18]	A bug in the propagation of 32-bit unsigned bounds from their 64-bit counterparts in the verifier enables out-of-bounds read/write in kernel memory.	✓
CVE-2020-8835 [17]	The verifier does not properly restrict the register bounds for 32-bit operations, which leads to out-of-bounds read/write in kernel memory.	✓
CVE-2020-27194 [16]	The verifier mishandles `scalar32_min_max_or` bounds-tracking during the use of 64-bit values, which allows out-of-bounds read/write in kernel memory.	✓

Table 4: A list of evaluated vulnerabilities.

non-trivial [56]. While we consider such an endeavor useful, it is beyond the scope of this work.

Fault Injection. To further demonstrate SafeBPF's effectiveness, we randomly inject out-of-bounds load and store operations to the output of the JIT compiler before SafeBPF's final binary rewriting step. This experiment simulates illegal memory accesses either undetected by the verifier or introduced during the JIT compilation. We repeat fault injections for 10, 000 times, and SafeBPF prevents *all* of them.

7 DISCUSSION & FUTURE WORK

Performance, Security, and Functionality. SafeBPF's primary objective is to allow developers to safely deploy unprivileged eBPF programs. The inability to do so in current Linux distributions is not only a functionality problem, but also a security one. We have witnessed a plethora of complex workarounds to run unprivileged programs in privileged mode by developers who are motivated by all the use cases for these programs. In a way, disabling unprivileged eBPF programs does not improve security, but on the contrary, increases the potential attack surface. While SafeBPF incurs performance overhead, it is a concrete step towards resolving this problem.

cgroups and Memory Tags. cgroups are used to organize and limit resource (e.g., CPU and memory) usage in a hierarchical fashion. They have gained popularity, alongside other namespaces, with the advent of containers. For example, in Kubernetes, the cgroup hierarchy is used to group containers logically. A number of eBPF programs, such as socket filters [13] and LSM-BPF [7, 79], can be associated (and only triggered) within a given cgroup and its descendants (see Kubernetes documentation [47]). One way to use SafeBPF is to associate sandboxes to cgroups. In case of MTE, this would mean associating a specific tag to each cgroup loading eBPF programs and maps. Doing so has two advantages. First, it guarantees no data leakage through eBPF programs across namespaces. Example use cases include individual containers safely deploying security audit policies [79] and scheduling policies [1, 2, 66]. Second, since MTE supports only 16 different tags, grouping eBPF programs and maps by cgroup would make efficient use of the limited available tags. Past work [82] has suggested a combination of MTE and ARM's pointer authentication feature [76] to significantly extend the number of isolation domains that can be created. Namespacing certain kernel features and their customization through eBPF requires careful consideration [79, 93] beyond the scope of this paper. We leave the exploration of such an approach to future work and conjecture that it might increase overhead.

Memory Tagging in Other CPU Architectures. Features in other CPU architectures, such as Intel PKS [98] and memory tagging in lowerRISC [43], provide functionalities similar to ARM MTE. We believe that the general design proposed in §4 should accommodate all of these architectures, even though the enforcement granularity may vary (e.g., PKS provides a 4KB page-size granularity). We intended to perform a comparison between Intel PKS and ARM MTE; however, issues around PKS availability made us reconsider (see more details in §D).

eBPF Tokens. Ongoing work [12] proposes to use eBPF tokens to manage eBPF program privileges. Simply put, eBPF tokens allow privileged processes to delegate privileges (e.g., access to certain eBPF program types or helper functions) to unprivileged processes. eBPF tokens are complementary to SafeBPF but orthogonal to the discussion of this paper.

Sandboxing and Speculative Execution. As discussed in §3, speculative execution vulnerabilities are out of scope in this paper. However, we note that sandboxing techniques, similar to those proposed here, have been presented as a potential solution to some speculative execution vulnerabilities [54, 84]. We leave the exploration of this topic to future work.

Software-based Isolation and Silent Failure. Sandbox enforcement through software-based isolation currently fails silently, because all invalid memory accesses are transformed into accesses inside a sandbox. Alternatively, we can add unmmaped guard zones surrounding a sandbox to increase the likelihood of faults upon out-of-bounds accesses [97].

Helper Functions and Access to Kernel Functions. As discussed in §3, SafeBPF relies on eBPF helper functions to perform safe operations. Indeed, there is no point in sandboxing eBPF programs, if attackers can simply use a helper function to modify credential data structures and give themselves root access. Accesses to helper functions are limited per eBPF program type. Helper functions accessible to unprivileged programs must be vetted extremely carefully.

We notice a growing trend where restrictions imposed on eBPF programs are being lifted (e.g., kfuncs [11] in which trusted eBPF programs can gain "raw" access to standard kernel functions). There is a spectrum of use cases for eBPF, from fully flexible customization of the kernel by a trusted party, to allowing a more restricted but safe program to be deployed by arbitrary applications. While the latter, closer to the original eBPF vision, is losing ground due to a large number of vulnerabilities discovered in the past few years, SafeBPF can help make it possible.

8 RELATED WORK

In-kernel Sandboxing. SafeBPF is an in-kernel sandboxing framework. Unlike SafeBPF, prior in-kernel sandboxing work [55, 58] focuses on *device drivers*, because they were (and continue to be) a major source of bugs in the Linux kernel. However, techniques to sandbox device drivers are not directly applicable to sandboxing eBPF programs. For example, microdrivers [60] and Decaf [89] partition a device driver's code into two components, migrating one that contains non-performance-critical code to userspace. As such, they leverage the hardware-enforced user/kernel privilege separation to "sandbox" the user-level component of a driver. However, since eBPF programs are often placed on critical paths, constantly accessing kernel data and calling kernel-level helper functions, the performance cost from context switching would be daunting. Later work [85–87] proposes to sandbox a driver in a separate virtual address space, but using similar techniques on eBPF programs would again incur high cost of switching between virtualization domains.

SafeBPF's software-based isolation approach has been used to sandbox device drivers [59, 81], but prior work often omits to check read instructions due to performance concerns [59, 81]. SafeBPF instruments all loads and stores to ensure both confidentiality and integrity of kernel memory.

HAKC [82] uses a combination of ARM's MTE and Pointer Authentication to sandbox device drivers, but it requires user annotations to specify isolation policy and marshal data across isolated compartments. SafeBPF instruments eBPF programs fully automatically, providing desirable security gurantees without increasing the burden for developers.

eBPF Sandboxing. SafeBPF is built upon SandBPF [77], with numerous extensions and improvements, including (1) introducing hardware-assisted isolation while SandBPF is only software-based; (2) simplifying access checking and thus reducing the number of additional instructions to be executed; (3) reducing context synchronization overhead by copying only data fields accessed by eBPF programs while SandBPF copies entire context objects; (4) protecting against type confusion vulnerabilities when calling helper functions while SandBPF does not; (5) supporting *any* eBPF programs, while SandBPF cannot run programs that use eBPF maps; and (6) being compatible with both x86 and ARM architectures.

Improving eBPF Security. Improving the security of eBPF programs is an active area of research. Using formal techniques to improve the quality of the built-in verifier [50, 95, 96] is one important line of work, while using fuzzing techniques [27, 30, 67, 74, 83]

to detect vulnerabilities in the eBPF framework is another. Jia et al. [68] recently proposed to move verification out of the kernel by leveraging Rust and a cryptographic signature scheme. These techniques are likely insufficient on their own, but they are complementary to SafeBPF, which provides defense-in-depth with a relatively low overhead.

9 CONCLUSION

We show that dynamic sandboxing improves memory safety of eBPF programs, complementary to the static mechanism employed by the eBPF verifier. While we do not foresee it replacing verification, it enhances the kernel's run-time safety, particularly when running unprivileged eBPF programs. Our framework, SafeBPF, implements dynamic sandboxing using both software-based and hardware-assisted approaches, imposing minimal overhead to make its adoption practical.

ACKNOWLEDGMENTS

We thank ATC 2024, EuroSys 2024 and CCSW 2024 reviewers for their help in improving the paper. 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 work was supported by Mitacs through the Mitacs Globalink Research Internship and the Globalink Graduate Fellowship programs. Cette recherche a reçu le soutien de Mitacs dans le cadre des programmes Stage de recherche Mitacs Globalink et Bourse aux cycles supérieurs Globalink.

REFERENCES

[1] 2021. ghOSt: Fast & Flexible User-Space Delegation of Linux Scheduling. https://lwn.net/Articles/873244/.
[2] 2022. eBPF Kernel Scheduling with Ghost. https://lpc.events/event/16/contributions/1365/attachments/986/1912/lpc22-ebpf-kernel-scheduling-with-ghost.pdf.
[3] 2024. Analysis on Kernel Self-Protection: Understanding Security and Performance Implication – Hardening Hostile Code in eBPF. Online (Accessed: 11th September 2024). https://samsung.github.io/kspp-study/bpf.html.
[4] 2024. /arch/arm64/kernel/cpufeature.c. Online (Accessed: 11th September 2024). https://elixir.bootlin.com/linux/v6.3.8/source/arch/arm64/kernel/cpufeature.c.
[5] 2024. Armv8.5-A Memory Tagging Extension. Online (Accessed: 11th September 2024). https://developer.arm.com/documentation/102925/latest/.
[6] 2024. bcc: Toolkit and library for efficient BPF-based kernel tracing. Online (Accessed: 11th September 2024). https://github.com/iovisor/bcc.
[7] 2024. bpf: cgroup_sock lsm flavor. Online (Accessed: 11th September 2024). https://lwn.net/Articles/899300/.
[8] 2024. BPF Compiler Collection (BCC). Online (Accessed: 11th September 2024). https://github.com/iovisor/bcc.
[9] 2024. BPF Exceptions. Online (Accessed: 11th September 2024). https://lwn.net/Articles/944372/.
[10] 2024. bpf: introduce BPF_JIT_ALWAYS_ON config. Online (Accessed: 11th September 2024). https://lore.kernel.org/lkml/20180109180429.1115005-1-ast@kernel.org/.
[11] 2024. BPF Kernel Functions (kfuncs). Online (Accessed: 11th September 2024). https://docs.kernel.org/bpf/kfuncs.html.
[12] 2024. BPF token. Online (Accessed: 11th September 2024). https://lwn.net/Articles/936823/.
[13] 2024. BPF_PROG_TYPE_CGROUP_SOCKOPT. Online (Accessed: 11th September 2024). https://docs.kernel.org/bpf/prog_cgroup_sockopt.html.
[14] 2024. The challenge of compiling for verified architectures. Online (Accessed: 11th September 2024). https://lwn.net/Articles/946254/.
[15] 2024. Cilium: eBPF-based Networking, Observability, Security. Online (Accessed: 11th September 2024). https://cilium.io/.
[16] 2024. CVE-2020-27194. Online (Accessed: 11th September 2024). https://nvd.nist.gov/vuln/detail/CVE-2020-27194.
[17] 2024. CVE-2020-8835. Online (Accessed: 11th September 2024). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8835.
[18] 2024. CVE-2021-31440. Online (Accessed: 11th September 2024). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-31440.
[19] 2024. CVE-2021-33200. Online (Accessed: 11th September 2024). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-33200.
[20] 2024. CVE-2021-34866. Online (Accessed: 11th September 2024). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-34866.
[21] 2024. CVE-2021-3490. Online (Accessed: 11th September 2024). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3490.
[22] 2024. CVE-2021-35039. Online (Accessed: 11th September 2024). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-35039.
[23] 2024. CVE-2021-4204. Online (Accessed: 11th September 2024). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-4204.
[24] 2024. CVE-2022-23222. Online (Accessed: 11th September 2024). https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-23222.
[25] 2024. CVE-2023-2163. Online (Accessed: 11th September 2024). https://nvd.nist.gov/vuln/detail/CVE-2023-2163.
[26] 2024. Falco: Cloud Native Runtime Security. Online (Accessed: 11th September 2024). https://falco.org/.
[27] 2024. Fuzzing for eBPF JIT bugs in the Linux kernel. Online (Accessed: 11th September 2024). https://scannell.io/posts/ebpf-fuzzing/.
[28] 2024. Guidance on Microsoft Signed Drivers Being Used Maliciously (Released: 13 Dec 2022 Last updated: 10 Jan 2023). Online (Accessed: 11th September 2024). https://msrc.microsoft.com/update-guide/vulnerability/ADV220005.
[29] 2024. Intel 64 and IA-32 Architectures Software Developer's Manual – Volume 1: Basic Architecture. Online (Accessed: 11th September 2024). https://cdrdv2.intel.com/v1/dl/getContent/671436.
[30] 2024. Introducing a new way to buzz for eBPF vulnerabilities. Online (Accessed: 11th September 2024). https://security.googleblog.com/2023/05/introducing-new-way-to-buzz-for-ebpf.html.
[31] 2024. Katran: A high performance layer 4 load balancer. Online (Accessed: 11th September 2024). https://github.com/facebookincubator/katran.
[32] 2024. Kernel Address Sanitizer (KASAN). Online (Accessed: 11th September 2024). https://docs.kernel.org/dev-tools/kasan.html.
[33] 2024. libbpf-bootstrap: demo BPF applications. Online (Accessed: 11th September 2024). https://github.com/libbpf/libbpf-bootstrap.
[34] 2024. Linux BPF Samples. Online (Accessed: 11th September 2024). https://github.com/torvalds/linux/tree/master/samples/bpf.
[35] 2024. LLVM improvements for BPF verification. Online (Accessed: 11th September 2024). https://lwn.net/Articles/974945/.
[36] 2024. Mitre: Rust CVEs. Online (Accessed: 11th September 2024). https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=rust.
[37] 2024. Netperf. Online (Accessed: 11th September 2024). https://hewlettpackard.github.io/netperf/.
[38] 2024. [PATCH V10 00/44] PKS/PMEM: Add Stray Write Protection. Online (Accessed: 11th September 2024). https://lore.kernel.org/lkml/20220419170649.1022246-1-ira.weiny@intel.com/.
[39] 2024. Pixie: Scriptable observability for Kubernetes. Online (Accessed: 11th September 2024). https://px.dev/.
[40] 2024. PKS: Add Protection Keys Supervisor (PKS) support. Online (Accessed: 11th September 2024). https://lwn.net/Articles/826091/.
[41] 2024. Runtime detection of CPU features on an ARMv8-A CPU. Online (Accessed: 11th September 2024). https://community.arm.com/arm-community-blogs/b/operating-systems-blog/posts/runtime-detection-of-cpu-features-on-an-armv8-a-cpu.
[42] 2024. Security Hardening: Use of eBPF by unprivileged users has been disabled by default. Online (Accessed: 11th September 2024). https://www.suse.com/support/kb/doc/?id=000020545.
[43] 2024. Tagged memory support. Online (Accessed: 11th September 2024). https://lowrisc.org/docs/tagged-memory-v0.1/tags/.
[44] 2024. Tigera: Container security with built-in network security. Online (Accessed: 11th September 2024). https://www.tigera.io/.
[45] 2024. Toward better handling of hardware vulnerabilities. Online (Accessed: 11th September 2024). https://lwn.net/Articles/764593/.
[46] 2024. Unprivileged eBPF disabled by default for Ubuntu 20.04 LTS, 18.04 LTS, 16.04 ESM. Online (Accessed: 11th September 2024). https://discourse.ubuntu.com/t/27047.
[47] 2024. Using eBPF in Kubernetes. Online (Accessed: 11th September 2024). https://kubernetes.io/blog/2017/12/using-ebpf-in-kubernetes/.
[48] Nikolaos Alexopoulos, Manuel Brack, Jan Philipp Wagner, Tim Grube, and Max Mühlhäuser. 2022. How Long Do Vulnerabilities Live in the Code? A Large-Scale Empirical Measurement Study on FOSS Vulnerability Lifetimes. In *Security Symposium (USENIX Sec'22)*. USENIX, 359–376.
[49] Maxime Belair, Sylvie Laniepce, and Jean-Marc Menaud. 2021. SNAPPY: programmable kernel-level policies for containers. In *Symposium on Applied Computing*. ACM, 1636–1645.

[50] Sanjit Bhat and Hovav Shacham. 2022. Formal Verification of the Linux Kernel eBPF Verifier Range Analysis.

[51] Marco Spaziani Brunella, Giacomo Belocchi, Marco Bonola, Salvatore Pontarelli, Giuseppe Siracusano, Giuseppe Bianchi, Aniello Cammarano, Alessandro Palumbo, Luca Petrucci, and Roberto Bifulco. 2022. hXDP: Efficient software packet processing on FPGA NICs. *Commun. ACM* 65, 8 (2022), 92–100.

[52] Xuechun Cao, Shaurya Patel, Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. 2024. FetchBPF: Customizable Prefetching Policies in Linux with eBPF. In *Annual Technical Conference (ATC'24)*. USENIX.

[53] 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.

[54] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. 2022. SoK: Practical foundations for software Spectre defenses. In *Symposium on Security and Privacy (S&P'22)*. IEEE, 666–680.

[55] Haogang Chen, Yandong Mao, Xi Wang, Dong Zhou, Nickolai Zeldovich, and M Frans Kaashoek. 2011. Linux kernel vulnerabilities: State-of-the-art defenses and open problems. In *Proceedings of the Second Asia-Pacific Workshop on Systems*. 1–5.

[56] Weiteng Chen, Xiaochen Zou, Guoren Li, and Zhiyun Qian. 2020. KOOBE: towards facilitating exploit generation of kernel Out-Of-Bounds write vulnerabilities. In *Security Symposium (USENIX Sec'20)*. USENIX.

[57] Ming-Chao Chiang, Tse-Chen Yeh, and Guo-Fu Tseng. 2011. A QEMU and SystemC-Based Cycle-Accurate ISS for Performance Estimation on SoC Development. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* (2011), 593–606.

[58] Andy Chou, Junfeng Yang, Benjamin Chelf, Seth Hallem, and Dawson Engler. 2001. An empirical study of operating systems errors. In *Proceedings of the eighteenth ACM symposium on Operating systems principles*. 73–88.

[59] 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.

[60] Vinod Ganapathy, Arini Balakrishnan, Michael M Swift, and Somesh Jha. 2007. Microdrivers: A new architecture for device drivers. *Network* 134 (2007), 27–8.

[61] Vinod Ganapathy, Matthew Renzelmann, Arini Balakrishnan, Michael Swift, and Somesh Jha. 2008. The design and implementation of Microdrivers. *ACM SIGARCH Computer Architecture News* (2008).

[62] 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 *Conference on Programming Language Design and Implementation (PLDI'19)*. ACM, 1069–1084.

[63] Yi He, Roland Guo, Yunlong Xing, Xijia Che, Kun Sun, Zhuotao Liu, Ke Xu, and Qi Li. 2023. Cross Container Attacks: The Bewildered {eBPF} on Clouds. In *32nd USENIX Security Symposium (USENIX Security 23)*. 5971–5988.

[64] 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 *International Conference on emerging Networking Experiments and Technologies (CoNEXT'18)*. ACM, 54–66.

[65] Yongzhe Huang, Vikram Narayanan, David Detweiler, Kaiming Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. 2022. KSplit: Automating Device Driver Isolation. In *Symposium on Operating Systems Design and Implementation (OSDI'22)*. USENIX, 613–631.

[66] Jack Tigar Humphries, Neel Natu, Ashwin Chaugule, Ofir Weisse, Barret Rhoden, Josh Don, Luigi Rizzo, Oleg Rombakh, Paul Turner, and Christos Kozyrakis. 2021. ghost: Fast & flexible user-space delegation of linux scheduling. In *Symposium on Operating Systems Principles*. ACM.

[67] Hsin-Wei Hung and Ardalan Amiri Sani. 2023. BRF: eBPF Runtime Fuzzer. *arXiv preprint arXiv:2305.08782* (2023).

[68] 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.

[69] Di Jin, Vaggelis Atlidakis, and Vasileios P Kemerlis. 2023. EPF: Evil Packet Filter. In *Annual Technical Conference (USENIX ATC'23)*. USENIX, 735–751.

[70] 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.

[71] Koen Koning, Xi Chen, Herbert Bos, Cristiano Giuffrida, and Elias Athanasopoulos. 2017. No need to hide: Protecting safe regions on commodity hardware. In *European Conference on Computer Systems (EuroSys'17)*. ACM, 437–452.

[72] Michael Larabel and Matthew Tippett. 2024. Phoronix test suite. Online (Accessed: 11th September 2024). *Phoronix Media* (2024). http://www.phoronix-test-suite.com.

[73] Dusol Lee, Inhyuk Choi, Chanyoung Lee, Sungjin Lee, and Jihong Kim. 2023. P2Cache: An Application-Directed Page Cache for Improving Performance of Data-Intensive Applications. In *Workshop on Hot Topics in Storage and File Systems (HotStorage'23)*. ACM, 31–36.

[74] Youlin Li, Weina Niu, Yukun Zhu, Jiacheng Gong, Beibei Li, and Xiaosong Zhang. 2023. Fuzzing Logical Bugs in eBPF Verifier with Bound-Violation Indicator. In *International Conference on Communications (ICC'23)*. IEEE, 753–758.

[75] Hans Liljestrand, Carlos Chinea, Rémi Denis-Courmont, Jan-Erik Ekberg, and N Asokan. 2022. Color My World: Deterministic Tagging for Memory Safety. *arXiv preprint arXiv:2204.03781* (2022).

[76] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N Asokan. 2019. PAC it up: Towards pointer integrity using ARM pointer authentication. In *Security Symposium (USENIX Sec'19)*. USENIX, 177–194.

[77] 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.

[78] Soo Yee Lim, Tanya Prasad, Xueyuan Han, and Thomas Pasquier. 2024. SafeBPF: Hardware-assisted Defense-in-depth for eBPF Kernel Extensions. In *Cloud Computing Security Workshop (CCSW'24)*. ACM.

[79] 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.

[80] Hongyi Lu, Shuai Wang, Yechang Wu, Wanning He, and Fengwei Zhang. 2023. MOAT: Towards Safe BPF Kernel Extension. *arXiv preprint arXiv:2301.13421* (2023).

[81] 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.

[82] 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.

[83] Mohamed Husain Noor Mohamed, Xiaoguang Wang, and Binoy Ravindran. 2023. Understanding the Security of Linux eBPF Subsystem. In *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems*. 87–92.

[84] Shravan Narayan, Craig Disselkoen, Daniel Moghimi, Sunjay Cauligi, Evan Johnson, Zhao Gang, Anjo Vahldiek-Oberwagner, Ravi Sahita, Hovav Shacham, Dean Tullsen, et al. 2021. Swivel: Hardening WebAssembly against spectre. In *Security Symposium (USENIX Sec'21)*. USENIX, 1433–1450.

[85] 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.

[86] 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.

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

[88] 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.

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

[90] R Sekar, Hanke Kimm, and Rohit Aich. 2024. eAudit: A Fast, Scalable and Deployable Audit Data Collection System. In *Symposium on Security and Privacy (S&P'24)*. IEEE.

[91] Farbod Shahinfar, Sebastiano Miano, Giuseppe Siracusano, Roberto Bifulco, Aurojit Panda, and Gianni Antichi. 2023. Automatic Kernel Offload Using BPF. In *Workshop on Hot Topics in Operating Systems (HotOS'23)*. ACM, 143–149.

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

[93] Yuqiong Sun, David Safford, Mimi Zohar, Dimitrios Pendarakis, Zhongshu Gu, and Trent Jaeger. 2018. Security namespace: making linux security frameworks available to containers. In *Security Symposium (USENIX Sec'18)*. USENIX, 1423–1439.

[94] 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.

[95] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2022. Sound, precise, and fast abstract interpretation with tristate numbers. In *International Symposium on Code Generation and Optimization (CGO'22)*. IEEE/ACM, 254–265.

[96] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2023. Verifying the Verifier: eBPF Range Analysis Verification. In *International Conference on Computer Aided Verification (CAV'23)*. Springer, 226–251.

[97] Robert Wahbe, Steven Lucco, Thomas E Anderson, and Susan L Graham. 1993. Efficient software-based fault isolation. In *Proceedings of the fourteenth ACM*

symposium on Operating systems principles. 203–216.

[98] Ira Weiny and Rick Edgecombe. 2024. Protection Keys, Supervisor (PKS). Online (Accessed: 11th September 2024). In *Linux Plumbers Conference*. Linux Foundation. https://lpc.events/event/11/contributions/907/attachments/787/1699/lpc-2021-PKS-22-Sept-2021.pdf.

[99] Emmett Witchel, Junghwan Rhee, and Krste Asanović. 2005. Mondrix: Memory isolation for Linux using Mondriaan memory protection. In *Symposium on Operating Systems Principles (SOSP'05)*. ACM.

[100] Zhe Yang, Youyou Lu, Xiaojian Liao, Youmin Chen, Junru Li, Siyu He, and Jiwu Shu. 2023. λ-IO: A Unified IO Stack for Computational Storage. In *Conference on File and Storage Technologies (FAST'23)*. USENIX, 347–362.

[101] Yuhong Zhong, Hongyi Wang, Yu Jian Wu, Asaf Cidon, Ryan Stutsman, Amy Tai, and Junfeng Yang. 2021. BPF for storage: an exokernel-inspired approach. In *Workshop on Hot Topics in Operating Systems (HotOS'21)*. ACM, 128–135.

[102] 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.

SUPPLEMENTARY MATERIAL

A SFI RESULTS

As shown in Table 5, our `sfi` implementation incurs 0%-4% overhead on the Apache and Nginx webserver benchmarks.

B MTE-MIN RESULTS

As shown in Table 6, our `mte-min` implementation incurs 0%-4% overhead on the Apache and Nginx webserver benchmarks.

C MODIFICATIONS TO TEST PROGRAMS

We modified `sockex1` and `sockex2` to use the helper function called `bpf_skb_load_bytes`, instead of the LLVM builtin functions (e.g., `load_bytes`, `load_half`, and `load_word`) to access network packets. All modified test programs will be made publicly available.

D PKS AVAILABILITY

We tested the presence of the PKS feature on a 13^{th} generation CPU (i9-13900K) and found that it was not supported. We checked Intel documentation [29], which states that the feature is present in "12^{th} generation Intel Core processor based on Alder Lake performance hybrid architecture [and] 4^{th} generation Intel Xeon Scalable Processor Family based on Sapphire Rapids microarchitecture". We contacted Intel and exchanged numerous e-mails and messages across several months and multiple channels. We were told that PKS is not supported in the 13^{th} generation and that "the technological feature has been removed from all core processors and Xeon products". We also note that patches to bring PKS support to the Linux [38, 40] have not been merged. The only mention in the Linux code base is the definition of `X86_FEATURE_PKS` as of kernel release 6.7-rc5. We could not find any official announcement. We refer interested readers to the work by Lu et al. [80].

E MTE SUPPORT ON APPLE M1 AND M2

Some sources on the internet state that Apple M1 and M2 CPUs are based on ARMv8.5A, but it does not appear to be the case or at least not all features are available or supported. First, we relied on Linux kernel CPU feature check [4] performed at boot time. MTE did not appear as an available feature. We then followed ARM instructions [41] to perform a sanity check, and confirmed that the feature was indeed unavailable. At the time of this writing, it is not clear if there is an open platform supporting MTE. Multiple online discussions seem to indicate that the latest Google Pixel 8 should support MTE, but the platform is closed. We are expecting accessibility to MTE to improve over the next few months or perhaps years. For example, Amazon has announced that its second-generation Neoverse CPU will support MTE. However, access to Neoverse V2 instances are currently restricted. We asked for early access, but our request was rejected at the time of this submission.

Test	sockfilter		sockex1		sockex2		dddos		vfs_read_lat	
	Vanilla	SafeBPF SFI	Vanilla	SafeBPF SFI	Vanilla	SafeBPF SFI	Vanilla	SafeBPF SFI	Vanilla	SafeBPF SFI
Requests per second (req/s)										
Apache 100	68808	67497 (**2 %**)	99391	96195 (**3 %**)	97782	95619 (**2 %**)	66868	64894 (**3 %**)	80745	79346 (**2 %**)
Apache 200	67862	67664 (**0 %**)	103741	103063 (**1 %**)	101212	100476 (**1 %**)	66770	66936 (**0 %**)	82804	79872 (**4 %**)
Apache 500	64510	64035 (**1 %**)	97828	97214 (**1 %**)	95597	93063 (**3 %**)	63170	60661 (**4 %**)	79111	77866 (**2 %**)
Apache 1000	63371	62250 (**2 %**)	95188	95482 (**0 %**)	94127	91378 (**3 %**)	64209	62482 (**3 %**)	78547	76197 (**3 %**)
Nginx 100	42376	41519 (**2 %**)	62514	61489 (**2 %**)	53609	52163 (**3 %**)	34615	34325 (**1 %**)	47831	46404 (**3 %**)
Nginx 200	42913	42431 (**1 %**)	62481	61548 (**1 %**)	53450	52068 (**3 %**)	35050	34964 (**0 %**)	47759	46015 (**4 %**)
Nginx 500	42057	41523 (**1 %**)	59563	58612 (**2 %**)	51174	50081 (**2 %**)	34895	34557 (**1 %**)	45896	44439 (**3 %**)
Nginx 1000	40804	40546 (**1 %**)	55778	54907 (**2 %**)	48063	46998 (**2 %**)	33470	33123 (**1 %**)	43224	41876 (**3 %**)

Table 5: Macrobenchmark measuring web server performance of SafeBPF SFI for 100-1000 concurrent connections.

Test	sockfilter		sockex1		sockex2		dddos		vfs_read_lat	
	Vanilla	SafeBPF MTE-min	Vanilla	SafeBPF MTE-min	Vanilla	SafeBPF MTE-min	Vanilla	SafeBPF MTE-min	Vanilla	SafeBPF MTE-min
Requests per second (req/s)										
Apache 100	68808	68645 (**0 %**)	99391	98758 (**1 %**)	97782	95907 (**2 %**)	66868	66314 (**1 %**)	80745	79037 (**2 %**)
Apache 200	67862	67395 (**1 %**)	103741	99957 (**4 %**)	101212	98406 (**3 %**)	66770	65202 (**2 %**)	82804	79549 (**4 %**)
Apache 500	64510	64827 (**0 %**)	97828	96679 (**1 %**)	95597	94498 (**1 %**)	63170	62743 (**1 %**)	79111	76833 (**3 %**)
Apache 1000	63371	63322 (**0 %**)	95188	94044 (**1 %**)	94127	91643 (**3 %**)	64209	63582 (**1 %**)	78547	75960 (**3 %**)
Nginx 100	42376	41571 (**2 %**)	62514	61148 (**2 %**)	53609	52758 (**2 %**)	34615	34109 (**1 %**)	47831	46604 (**3 %**)
Nginx 200	42913	42389 (**1 %**)	62481	61251 (**2 %**)	53450	52700 (**1 %**)	35050	34482 (**2 %**)	47759	46549 (**3 %**)
Nginx 500	42057	42046 (**0 %**)	59563	58651 (**2 %**)	51174	50740 (**1 %**)	34895	34596 (**1 %**)	45896	44737 (**3 %**)
Nginx 1000	40804	40722 (**0 %**)	55778	54842 (**2 %**)	48063	47612 (**1 %**)	33470	33122 (**1 %**)	43224	42223 (**2 %**)

Table 6: Macrobenchmark measuring web server performance of SafeBPF MTE-min for 100-1000 concurrent connections.