MOAT: Towards Safe BPF Kernel Extension

Hongyi Lu, *Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, and Hong Kong University of Science and Technology;* Shuai Wang, *Hong Kong University of Science and Technology;* Yechang Wu and Wanning He, *Southern University of Science and Technology;* Fengwei Zhang, *Southern University of Science and Technology and Research Institute of Trustworthy Autonomous Systems*

https://www.usenix.org/conference/usenixsecurity24/presentation/lu-hongyi

This paper is included in the Proceedings of the 33rd USENIX Security Symposium.

August 14–16, 2024 • Philadelphia, PA, USA

978-1-939133-44-1

Open access to the Proceedings of the 33rd USENIX Security Symposium is sponsored by USENIX.

MOAT: Towards Safe BPF Kernel Extension

Hongyi Lu[1,2,3], Shuai Wang[3,†], Yechang Wu[2], Wanning He[2], Fengwei Zhang[2,1,†]

[1]*Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology*
[2]*Department of Computer Science and Engineering, Southern University of Science and Technology*
[3]*Department of Computer Science and Engineering, Hong Kong University of Science and Technology*

Abstract

The Linux kernel extensively uses the Berkeley Packet Filter (BPF) to allow user-written BPF applications to execute in the kernel space. The BPF employs a verifier to check the security of user-supplied BPF code statically. Recent attacks show that BPF programs can evade security checks and gain unauthorized access to kernel memory, indicating that the verification process is not flawless. In this paper, we present MOAT, a system that isolates potentially malicious BPF programs using Intel Memory Protection Keys (MPK). Enforcing BPF program isolation with MPK is not straightforward; MOAT is designed to alleviate technical obstacles, such as limited hardware keys and the need to protect a wide variety of BPF helper functions. We implement MOAT on Linux (ver. 6.1.38), and our evaluation shows that MOAT delivers low-cost isolation of BPF programs under mainstream use cases, such as isolating a BPF packet filter with only 3% throughput loss.

1 Introduction

It is common to extend kernel functionality by allowing user applications to download code into the kernel space. In 1993, the well-known Berkeley Packet Filter (BPF) was introduced for this purpose [7]. The classic BPF is an infrastructure that inspects network packets and decides whether to forward or discard them. With the introduction of its extended version (referred to as eBPF) in the Linux kernel, BPF soon became more powerful and is now utilized in numerous real-life scenarios, such as load balancing, system tracing, and system call filtering [24, 30, 37, 63, 71, 72].

To ensure security, BPF is equipped with a *verifier* [9]. The verifier performs a variety of static analyses to ensure the user-supplied code is secure. For instance, the verifier tracks the bounds of all pointers to prevent out-of-bound access. Given that BPF code runs directly within the kernel, the verifier becomes crucial for BPF security. Nevertheless, as pointed out by recent studies [32, 43, 44, 61, 69], the current verifier has various limitations and is insufficient for the overall security of BPF. First, the current BPF ecosystem supports a variety of functionalities, such as packet forwarding and kernel debugging [10, 31]. Supporting all these functionalities in the verifier results in a complicated verification process. Though the verifier has been partially verified via formal methods [68], the unverified part and the gap between abstraction and implementation still result in vulnerabilities [47–51, 53, 54, 56]. Second, due to the rapid expansion of BPF capabilities, the verifier is frequently updated, and it is inherently difficult to update a complex static verification tool without introducing new vulnerabilities [55]. To date, the BPF subsystem has been repeatedly exploited. For instance, two privilege-escalation vulnerabilities have been discovered in `bpf_ringbuf`, a recent BPF feature introduced in 2020 [7]. Further, the verifier's register-value tracking is quite complex and often bypassed via corner-case operations (e.g., sign extension) [47–50].

Given the increasing security threats in BPF and the challenge of enforcing safe BPF programs with merely static verification, we seek to employ hardware extensions to sandbox untrusted BPF programs. In particular, we leverage Intel Memory Protection Keys (MPK) [11], an emerging hardware extension that partitions memory into distinct permission groups by assigning up to 16 keys to their Page Table Entries (PTEs). With the aid of MPK, we present MOAT, which isolates untrusted BPF programs in a low-cost and principled manner. For instance, two MPK protection keys K and E can be assigned to the kernel and the BPF programs, respectively. When the kernel transfers control to a BPF program, it can set K as access-disabled to prevent the potentially malicious BPF program from tampering with kernel memory.

Despite its promising potential, using MPK to enforce BPF isolation is not straightforward. In designing MOAT, we faced and overcame *two major technical hurdles*. First, MPK provides a maximum of 16 keys. Thus, supporting numerous BPF programs with this limited number of keys is challenging. Existing workarounds like key virtualization [62] heavily rely on scheduling and notification mechanisms that are only available to user space; our tentative observation shows that

† Shuai Wang and Fengwei Zhang are the corresponding authors.

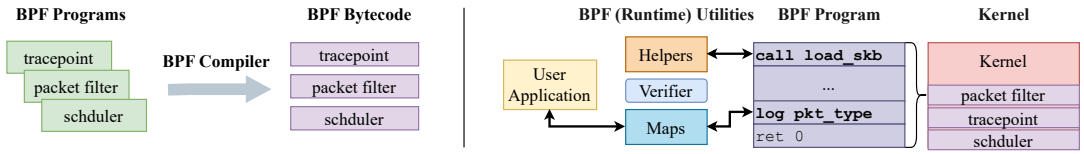

Figure 1: BPF overview. We illustrate the BPF compilation procedure and execution context of a sample BPF packet filter.

directly reusing them in the kernel may largely block kernel threads. To address this hurdle, we propose a novel two-layer isolation scheme to protect both the kernel and the benign BPF programs from malicious BPF programs. MOAT also utilizes a contemporary hardware feature named process context identifier to minimize the incurred overhead. Second, while MPK-based isolation mitigates malicious BPF programs, *helper functions* provided by the BPF subsystem may still be exploited by attackers. On the one hand, MOAT should allow benign BPF programs to use these helpers freely. On the other hand, MOAT must be cautious enough with these helpers to ensure that they are not abused. However, designing security policies for each of them requires non-trivial engineering effort and might result in a bloated codebase. To prevent abuse, we design two defense schemes that do not rely on the specific design of helper functions. We show that each of them applies to a wide range of helpers (see Appendix. C).

We systematically examine how MOAT mitigates the attack on the BPF ecosystem and the potential threats to MOAT itself. We also empirically analyze all recent CVEs within MOAT's application scope. The result shows that MOAT successfully mitigates each CVE. We evaluate the performance overhead brought by MOAT across a variety of micro and macro benchmark settings, and MOAT achieves low performance overhead across all settings. In particular, we evaluate MOAT with the common use cases of BPF in network applications, and the maximum performance penalty from MOAT is 3% among these cases. We also test MOAT's overhead on system tracing, another important BPF use case. On average, MOAT brings a performance loss of 5.5% in this setting. Furthermore, we evaluate MOAT's performance when using BPF for system call filtering [12]. In this case, the performance loss brought by MOAT is less than 3%. Thus, we conclude that MOAT's overhead is reasonably low, especially given its security benefits. In sum, we have made the following contributions.

- Instead of merely relying on the BPF verifier to statically validate BPF programs, this paper, for the first time, advocates isolating BPF programs with an emerging hardware extension, Intel MPK, effectively ensuring the memory safety of BPF programs.
- Technically, MOAT is properly designed to address domain-specific challenges, including limited hardware keys and preventing helper abuse in the BPF ecosystem. MOAT features a two-layer isolation scheme to protect both the kernel and the benign BPF programs from malicious BPF programs and incorporates various design considerations to deliver a security guarantee on memory safety at a low cost.
- We implemented a prototype of MOAT on Linux 6.1.38[1] and thoroughly evaluated its security over different attack scenarios (including all memory-relevant BPF CVEs in the past decade) and performance using various benchmark datasets. The evaluation shows that MOAT delivers a principled security warranty with minimum overhead.

2 Background

2.1 Berkeley Packet Filter (BPF)

BPF Overview. BPF [7] was originally introduced to facilitate flexible network packet filtering. Instead of inspecting packets in the user space, users can provide BPF instructions specifying packet filter rules, which are directly executed in the kernel. BPF allows configurable packet filtering without costly context switching and data copying. Modern Linux kernel features extended BPF (eBPF), a Linux subsystem which supports a wide range of use cases, such as kernel profiling, load balancing, and firewalls. Popular applications such as Docker [46], Katran [31], and kernel debugging utilities like Kprobes [10] utilize or are built directly on top of BPF.

Fig. 1 depicts an overview of how BPF programs are compiled and deployed. The BPF subsystem offers ten general-purpose 64-bit registers, a stack, BPF customized data structures (often called BPF maps), and a set of BPF helper functions. To use BPF (e.g., for system tracing), users first write their own BPF programs (in C code) to specify the functionality, which, in turn, are compiled into bytecode and loaded into the kernel. Given that BPF code is written by untrusted users, the kernel employs a verifier to conduct several checks during the bytecode loading stage (see below). By default, the verified bytecode is further compiled into native code by an in-kernel Just-In-Time (JIT) compiler for better performance. Additionally, on platforms without the JIT support, the bytecode is alternatively executed by the BPF interpreter. The BPF program is then attached to certain kernel components based on its specific end goal. For instance, as shown in Fig. 1, a BPF program attaches to the kernel as the packet filter, monitoring network traffic and sending statistics back to the user space via a BPF map.

BPF Verifier. BPF programs are written in C and compiled into a RISC-like instruction set. As aforementioned, the kernel

[1]We release the codebase of MOAT on our site [14]. We will maintain MOAT to benefit the community and follow-up research.

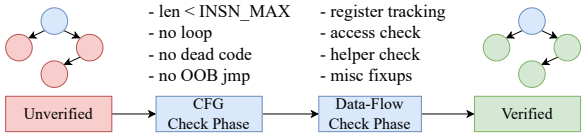

Figure 2: BPF verification process.

strictly verifies the BPF programs upon loading to ensure they are safe to execute. Fig. 2 illustrates the verification process in a holistic manner. First, a BPF program is parsed into a control flow graph (CFG) by the verifier, which performs a CFG check phase to ensure four key properties: 1) the program size is within a limit; 2) there are no back edges (loops) on its CFG; 3) there is no unreachable code; and 4) all jumps are direct jumps and refer to a valid destination.

The verifier then tracks the value flow of every register to deduce its value ranges conservatively. With these ranges, the verifier decides if a pointer accesses safe memory and if a parameter is valid. Since this analysis is performed statically, it is possible for a malicious BPF program to exploit a vulnerability to bypass it [47–51, 53, 54, 56].

BPF Helpers. The kernel also limits the functions a BPF program may call. Those functions are dubbed BPF helpers, as shown in Fig. 1. To date, there are over 200 helpers provided by the kernel [4]. Depending on the task, a BPF program can usually call a group of relevant helpers. For example, a BPF packet filter can call `skb_load` to read packet data, but is not allowed to call any helper related to system tracing.

BPF Maps. Out of security concern, the kernel also sets a strict space limit on BPF programs. Each program, by default, can only use up to 512 bytes of stack space and 10 registers, which is far from enough for certain BPF programs. To address this problem, BPF maps can be allocated to provide additional space for BPF programs. To date, there are over 30 types of maps supported by kernel [8]. Based on the isolation requirements, they can be roughly categorized into two types. The first type is maps that own a memory region. The most commonly used maps, hash maps and array maps, belong to this category. BPF programs use them to store data and communicate with user space. Therefore, a proper access permission has to be set for these maps (see MOAT's solution in Sec. 4.1). The second type holds references to other kernel resources (e.g., file descriptors). BPF programs are restricted to using helpers to interact with this type of map. Thus, MOAT forbids BPF programs from directly accessing them.

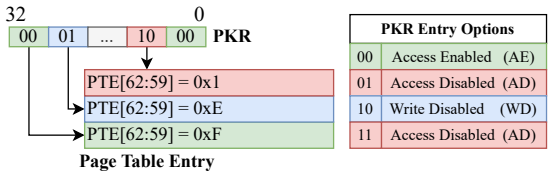

Figure 3: Intel MPK overview.

Classic BPF (cBPF). cBPF specializes in tasks like syscall filtering (e.g., `seccomp-BPF`) and has more restrictions than eBPF. We clarify that MOAT supports both of them. We also evaluate MOAT using `seccomp-BPF` with cBPF in Sec. 6.2.2. In this paper, we use BPF to refer to both cBPF and eBPF, as the kernel internally converts cBPF to eBPF.

2.2 Hardware Features in MOAT

Intel MPK. Intel introduced MPK [11] to provide efficient page table permissions control. By assigning an MPK protection key to the page table entries (PTEs) of one process, users can enable intra-process isolation and confidential data access control [23, 45, 62, 67]. As illustrated in Fig. 3, MPK uses four reserved bits `[62:59]` in each PTE to indicate which protection key is attached to this page. Those three PTEs in Fig. 3 are assigned with keys `0x1`, `0xE` and `0xF`, respectively. Since there are only 4 bits involved, the maximum number of keys is 16. Then, a new 32-bit register named Protection Key Register (PKR) is introduced to specify the access permission of these protection keys. Each key occupies two bits in PKR, whose values flag the access permission of the page. In Fig. 3, the access permissions of the three pages are `01` access-disabled (AD), `10` write-disabled (WD), and `00` access-enabled (AE), respectively. By writing to certain bits in PKR, the access permission of corresponding pages can be configured efficiently without having to modify the PTEs.

Clarification and Notations. There are actually two versions of MPK. One applies to the user space, while the other applies to the kernel space. For brevity, we refer to these two versions in their conventional abbreviations as Protection Key Supervisor (PKS) and Protection Key User (PKU), respectively. Most existing works [23, 34, 45, 62, 67] are based on PKU. In MOAT, we use PKS instead since our goal is to isolate in-kernel BPF programs. The logistics behind these two versions are mostly identical with slight variations. For instance, the permission configuration register in PKS is a Model Specific Register (MSR) named `IA32_PKRS`, which is inaccessible from user space, whereas in PKU, this role is assigned to a dedicated register `PKRU`. To avoid confusion, the rest of the paper refers to MPK leveraged by MOAT as PKS.

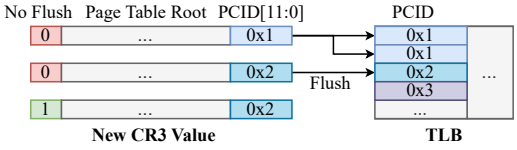

Figure 4: PCID overview.

Process Context Identifier (PCID). MOAT uses PCID to reduce the overhead of address-space switching (Sec. 4.2.2); we introduce PCID here. On the x86 platform, the CR3 register holds the page table root of the current process. Modifying the CR3 register causes a complete Translation Lookaside Buffer (TLB) flush and is therefore costly. Fortunately, Intel introduced PCID to address this issue. As shown in Fig. 4, the lower 12 bits `[11:0]` of CR3 register are PCID, identify-

ing the owner of the page table, while the highest bit of the new CR3 value controls the flushing behavior of TLB. If the highest bit is 1, this modification does not flush TLB at all; if the highest bit is 0, then this modification only flushes the TLB entries of the PCID in this new CR3 value. This feature enables fast address-space switch without costly TLB flush. Since there are only 12 reserved bits for PCID, it supports up to 4096 different processes with isolated TLB entries.

3 Motivation and Threat Model

3.1 Motivation

In this section, we discuss the typical threats to the BPF verifier, the restriction on unprivileged BPF brought by these threats, and lastly, the motivation for our research.

Fast Feature Evolving. As a fast-developing technology, threats may come from the inconsistency between the constantly expanding BPF capabilities and the rigorous static verification process imposed on them [51, 55]. It is a common practice to add corresponding verification procedures simultaneously when introducing new features to BPF programs. However, it is difficult to implement a verifier that supports all these features yet still does not miss any edge case, which already has over 10K LoC with various functionalities [9].

Challenging Register-value Tracking. Second type of threats originates from the complexity of the register-value tracking. Although the soundness of such a tracking mechanism is formally proved [68], there exist gaps between the actual implementation and abstraction of the register-value tracking, especially in some corner cases, such as sign extension, truncation, and bit operators [47–51, 53, 54, 56].

Unprivileged BPF. BPF was originally designed as a restricted interface for *unprivileged* users to extend kernel functionality. It comes with a fine-grained privilege system [6] that allows users to tune a specific part of the kernel without root. However, numerous vulnerabilities [47–50] indicate that the verifier is not reliable, and consequently, major distributions have banned unprivileged users from loading BPF programs [38, 65]. Despite this, there is still a long-lasting desire for unprivileged BPF in the community. For example, the `seccomp-BPF` users have been asking for unprivileged BPF support for a long time [5].[2] Moreover, there have been continuous efforts (from 2016 to 2023) in the community to re-emerge unprivileged BPF again [1, 3, 13]. Unfortunately, these efforts fail as they only focus on enhancing the verifier itself, which is already over-complicated and error-prone.

Motivation. Overall, seeing BPF's potential and its current restriction, we propose MOAT as an isolation scheme complementary to the BPF verifier. On the one hand, this isolation scheme shall make BPF more accessible to unprivileged users whilst maintaining security. On the other hand, even for privileged users, MOAT provides the security guarantee that the BPF programs obtained from a potentially untrusted source are isolated from the kernel. Overall, we aim to provide a more secure and accessible BPF ecosystem, thereby promoting its development and adoption in the community.

[2] We clarify that `seccomp` already supports classic-BPF (cBPF), which lacks expressiveness and no longer updates [2]. BPF here refers to eBPF.

3.2 Threat Model

Our threat model considers a practical setting that is aligned with existing BPF vulnerabilities [47–51, 53, 54, 56]. Attackers can load their prepared BPF code into the kernel space to launch exploitation. In particular, we assume attackers are *non-privileged users* with BPF access since a root user already has control over almost the entire kernel. MOAT isolates user-submitted BPF programs and prevents them from accessing kernel memory regions. As will be introduced in Sec. 4, a BPF program is given only the necessary resources and privileges to complete its task. We present the threat models of major components in our research context as follows.

BPF Programs. We assume that malicious BPF programs are able to bypass checks statically performed by the verifier; they may thus behave maliciously during runtime. Our threat model deems BPF programs as *untrusted*.

BPF Helper Functions. These helpers act as the intermediate layer between the BPF subsystem and kernel. Certain malicious BPF programs can abuse these helpers to perform attacks, and therefore, we assume they are also *untrusted*. MOAT mitigates risks raised by adversarial-manipulated helper functions with practical defenses.

Out of Scope. The main objective of MOAT is to mitigate memory exploitation performed by BPF programs. Other subtle attacks (not relevant to memory exploits), such as speculation, race condition, and Denial of Service (DoS) toward the BPF subsystem [57, 58] are not considered. They do not specifically exist in BPF [22, 26], and are addressed by relevant research [21, 29]. We thus treat them as orthogonal. Also, BPF subsystem comes with a set of user-space facilities such as `libbpf`; bugs in them are not considered by MOAT. Note that MOAT mitigates information leakage that is due to out-of-bounds memory access; if the leakage is due to issues like speculation [58], then it is out of the scope of MOAT.

We clarify that MOAT focuses on the kernel memory exploitation via BPF, its most prevalent threat. The vulnerabilities mitigated by MOAT typically receive high threat scores in vulnerability databases [47–51, 53–56] with public PoC exploits [66], whereas above-precluded vulnerabilities often lack exploits [27].

4 Design

MOAT Overview. As described in Sec. 3.1, the current security design against malicious BPF programs solely relies on the static analysis performed by the BPF verifier, which

is seen as a weak point and exploitable by non-privileged users. MOAT instead delivers a principled isolation of BPF programs from the rest part of the kernel using PKS and prevents bypasses.

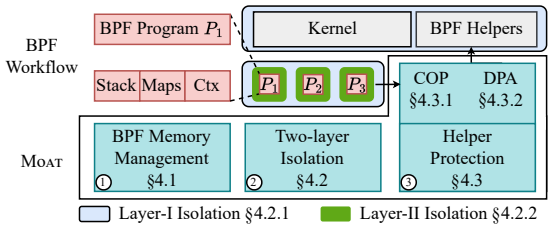

Figure 5: MOAT overview.

Fig. 5 depicts an overview of MOAT and how it is integrated into the workflow of BPF programs. ① Given a user-submitted BPF program *P*, MOAT statically allocates the necessary memory regions the program needs, such as stack, maps, and context based on *P*'s metadata (Sec. 4.1). ② When the kernel invokes *P*, MOAT isolates *P* from the kernel using PKS (Layer-I in Sec. 4.2.1), and constrains *P* in its isolated address space (Layer-II in Sec. 4.2.2). ③ On the occasions that *P* calls helpers, depending on the helper types, MOAT adjusts the involved memory region permissions (Sec. 4.3.1) and validates the helper parameters (Sec. 4.3.2) to prevent the helpers from being abused.

4.1 BPF Memory Management in MOAT

Further to the overview in Fig. 5, we introduce how MOAT manages the BPF memory. The BPF memory refers to the memory regions a BPF program needs to function properly, including descriptor tables, stacks, maps, and runtime context.

Descriptor Tables. On x86 platforms, Global Descriptor Table (GDT) and Interrupt Descriptor Table (IDT) are essential for basic operations like interrupt. These structures are assigned to a shared region that all BPF programs can access. To prevent tampering, they are made read-only when shared.

Stack. BPF programs use a 512-byte stack space to store local variables and function frames. The verifier determines if a program makes out-of-bounds access toward the stack. Thus, if the BPF program passes the static checks, its stack is directly allocated from the kernel stack. However, as discussed in Sec. 3.1, certain vulnerabilities may allow BPF programs to bypass this check. Thus, MOAT needs to allocate the stack as a part of BPF memory and swap stacks to prevent the BPF programs from tampering with the kernel stack.

Maps. As described in Sec. 2.1, maps are utilized by BPF programs to store data and communicate with the user space. Linux provides a set of helper functions for BPF programs to interact with maps. For example, `bpf_map_look_up_elem` returns the pointer of an element so that the program can modify its value. This means that BPF programs must have access to these elements' memory. Thus, MOAT allocates these maps as a part of BPF memory. Note that we do not allocate the metadata of these maps inside BPF memory since they contain exploitable structures like function pointers.

Runtime Context. The context refers to BPF program parameters, which vary depending on the BPF program types. We investigated the BPF contexts of common BPF program types and summarized our findings in Table 1. Most of these contexts are local objects on the kernel stack and are passed to BPF programs as parameters, such as `bpf_cgroup_dev_ctx`. For this type of BPF context, MOAT allocates them on the BPF stack instead so that the BPF programs can still access them without the permission to access the kernel stack. However, there also exist contexts that are not local objects on the stack but persistent kernel structures. For example, `sk_buff` holds the network packet received by a socket and is also passed to BPF socket filter programs as context. For this type of persistent context (denoted in the fourth column of Table 1), MOAT dynamically maps the physical page of the corresponding context into the BPF memory. The reason why we choose to map instead of creating a local copy is that `sk_buff` is typically hundreds of bytes. Our preliminary experiment shows that syncing between the local copy and the actual kernel object brings non-trivial overheads. Furthermore, network-related BPF contexts (e.g., `bpf_sock_ops`) may contain nested pointers to other kernel structures (denoted in the fifth column of Table 1). Including only these pointers in BPF memory triggers a false alarm, as these nested structures are not included. We clarify that BPF programs only access limited fields of these nested structures. Thus, MOAT reserves a part of BPF memory to mirror these nested fields efficiently so that they can be accessed by the BPF programs.

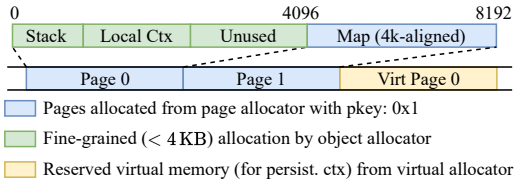

Figure 6: BPF memory allocators.

BPF Memory Allocators. As shown in Fig. 6, MOAT provides three types of allocators to manage these BPF memory regions. When loading a BPF program, the page allocator first allocates physical pages (Page 1) for its BPF memory; these pages are given the protection key `0x1` and become a part of its BPF memory. The object allocator handles fine-grained allocations from the allocated pages (Page 0) that are less than the page size (4 KB), e.g., the BPF stacks (512 bytes). Lastly, the virtual allocator controls the virtual memory that is not backed with concrete physical pages. For instance, it reserves a part of virtual BPF memory (Virt Page 0) to map persistent BPF contexts (e.g., `sk_buff`). Note that we also modify the implementation of BPF maps to use MOAT's allocator.

Table 1: BPF context of common program types.

Category	Program Type	Context Type	Persistent	Nested	Note
Network	Socket Filter	`sk_buff *`	Yes	Yes	Socket packet buffer
	Socket Ops	`bpf_sock_ops *`	No	Yes	Socket events (timeout, retransmission, ...)
	Socket Lookup	`bpf_sk_lookup *`	No	Yes	Packet information for socket lookup
	XDP	`xdp_md *`	No	Yes	Metadata of `xdp_buff`
Tracing	Kprobe	`pt_regs *`	No	No	Register status on probed location
	Tracepoints	Depending on tracepoint types	No	No	Relevant tracepoint information
	Perf Event	`bpf_perf_event_data *`	No	No	Perf. event (register status, sample period)
Cgroup	Cgroup Socket Filter	`sk_buff *`	Yes	Yes	Socket packet buffer under specific cgroup
	Cgroup Device	`bpf_cgroup_dev_ctx *`	No	No	Device ID, access type (read, write)

4.2 Two-layer Isolation

Challenge. In theory, we can assign each BPF program with a unique key to achieve low-cost BPF isolation. However, PKS only supports up to 16 regions (i.e., keys). If we assign each BPF program with a unique key, these keys would soon be exhausted as there could be over 16 BPF programs in the kernel. It is challenging to isolate an unlimited number of BPF programs with only 16 protection keys.

Solution. We propose a novel two-layer isolation scheme using PKS and isolated address spaces. Though isolating address spaces is less efficient than PKS, we manage to reduce its overhead to a minimum using a contemporary hardware feature named PCID; We use PCID as a complement to PKS to support the isolation of numerous BPF programs.

4.2.1 Layer-I: Lightweight Isolation Domain via PKS

The main objective of MOAT is to isolate the kernel from malicious BPF programs. Thus, we use PKS as a lightweight isolation primitive between kernel and BPF programs. Specifically, we use PKS to build three isolated domains: the BPF domain, the kernel domain, and the shared domain.

As depicted in Fig. 7, all BPF programs reside in the BPF domain with the protection key `0x1`. MOAT grants a BPF program access (i.e., access-enabled; AE) to the BPF domain (`0x1`) when executing the program by setting its PKR bits to `00` (flagging AE). The kernel domain holds all kernel pages with the protection key `0x0` and is only accessible by the kernel itself. When entering a BPF program, this kernel domain (`0x0`) becomes access-disabled (AD) by setting its PKR bits to `01` (flagging AD). However, the shared domain (`0x2`) comprises memory regions like IDT and GDT. These regions are crucial for low-level routines like interrupts. Thus, they are made write-disabled (WD) instead of access-disabled for BPF programs by setting the PKR bits to `10` (flagging WD).

This domain design only needs three (out of 16) keys from PKS yet effectively mitigates malicious BPF programs targeting the kernel. For these malicious programs, a modus operandi is to introduce an unsanitized kernel pointer by exploiting a verifier vulnerability. Then, the malicious BPF program arbitrarily tampers the kernel using that pointer, leading to a full-blown exploitation. Isolating BPF programs from the kernel effectively stops such attacks, as all malicious kernel access directly from BPF programs is prevented by PKS.

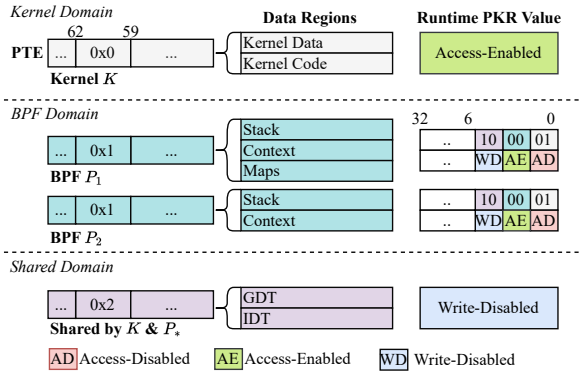

Figure 7: PKS-enforced domains of MOAT.

4.2.2 Layer-II: Isolated BPF Address Space

In Sec. 4.2.1, we have discussed how MOAT prevents BPF attacks targeting the kernel. However, MOAT only uses PKS to build isolation between the kernel and BPF programs. All BPF programs still share the same PKS domain, allowing a malicious BPF program to tamper with the memory of benign BPF programs. Inspired by prior works in user-space isolation [70], we set up an isolated address space for each BPF program to prevent such tampering. Consequently, when a malicious BPF program tries to access the memory regions of another BPF program, a page fault occurs, and the malicious BPF program is immediately terminated.

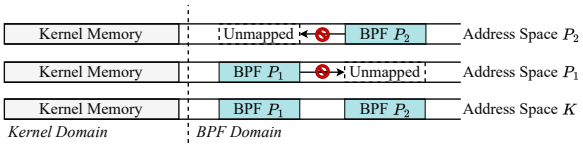

Figure 8: Isolated address spaces of BPF programs.

Fig. 8 illustrates the isolated address spaces of two BPF programs, P_1 and P_2. In the address space of P_1, the mapping of P_2 does not exist. Similarly, the mapping of P_1 does not exist in the address space of P_2 either. This effectively prevents BPF programs from accessing each other and addresses the abovementioned issue. To avoid the high TLB flush overhead (and TLB misses) from the address-space switching, we use PCID to keep the TLB entries from different BPF programs isolated. Since BPF programs are usually smaller than user applications, MOAT allocates a non-overlapped virtual address space with a unique PCID for each of them. In the rare cases where over 4,096 BPF programs are running in the

same kernel, MOAT has to flush the TLB when two BPF programs (with the same PCID) run consecutively. Since there are 4,096 PCIDs available, we expect such conflicts to be rare, especially considering that the kernel also periodically flushes TLB, thus clearing the conflicting entries. Even when such cases occur, MOAT only flushes the TLB entries of the conflicting PCID and leaves other TLB entries intact.

Why Intra-BPF Isolation. One may question the necessity for the isolation between BPF programs, as most existing BPF exploits target the kernel. However, since BPF maps are the only bridge between the BPF programs and the user space, the configurations of a BPF program have to be saved in its maps so that users can change its behavior without reloading it. This paradigm makes cross-BPF attack a noticeable threat [17]. For example, an attacker may load a malicious BPF program to change the behavior of another program by tampering with its configuration maps, disrupting resources accounting, or even nullifying security checks. Intra-BPF isolation is essential for preventing such attacks.

4.3 Helper Security Mechanism

As mentioned in Sec. 2.1, the kernel provides a set of helper functions for BPF programs. As these helpers act as the interfaces between the kernel and BPF programs, they can also be abused by malicious programs to launch attacks. MOAT needs to protect the helpers from such abuse.

Our investigation of existing BPF vulnerabilities shows that the malicious BPF programs typically abuse the BPF helpers in two ways: ① the helper contains a defect, which is exploited by the malicious programs [52]; ② the helper itself is correct, but the malicious programs pass invalid parameters to abuse it [55]. For ①, a typical case is that the helper itself contains defects such as heap overflow. These defects are leveraged by malicious programs to overwrite the sensitive fields (e.g., function pointers) of BPF-related kernel objects. For ②, since the helper by itself is correct, the malicious programs are typically restricted to leaking kernel pointers (by passing invalid parameters) and cannot conduct full-blown exploitation. Notably, in most cases, since the helper parameters are checked by the BPF verifier, the malicious programs still need to leverage the register-value-tracking vulnerabilities in the verifier (Sec. 3.1) to bypass this check [47–51, 53, 54, 56].

Challenge. However, protecting these BPF helpers is not trivial. First, the BPF helpers, by design, need to access BPF-related objects in the kernel memory; blindly isolating helpers using PKS leads to spurious alarms and impedes benign programs. Second, there are over 200 BPF helpers in the kernel, so MOAT's design must be generic enough to apply to most of these helpers (see Appendix. C for the supported helpers).

Design Consideration and Solution. To prevent such abuse, we aim to identify and guard sensitive BPF-related objects (instead of all BPF-related objects) from the defective BPF helpers. Besides directly protecting sensitive objects, since the attackers need to deliver malformed parameters to conduct helper abuse, we also wish to ensure the validity of the helper parameters at runtime. To this end, we designed the following two defense schemes: Critical Object Protection (COP) and Dynamic Parameter Auditing (DPA). COP protects sensitive BPF-related objects from being tampered with, while DPA dynamically checks if the arguments of the helpers are within legitimate ranges. To clarify, COP and DPA should be enabled together to deliver protection. DPA only constrains the arguments to their expected ranges. This stops most exploitation attempts [47–51] but may not ward them off completely in the presence of a buggy helper [52]; COP, in this case, prevents the buggy helper from accessing sensitive objects.

4.3.1 Critical Object Protection (COP)

Although BPF helpers have to access BPF-related kernel objects to complete their tasks, the sensitive BPF-related objects should still not be accessed by any helper. For example, `array_map_ops` is a function pointer in the BPF array maps that should *only* be accessed from system calls. However, `array_map_ops` is close to other helper-needed objects in the address space, making it a potential victim of the abused helpers. Based on this, we designed the COP scheme. As shown in Fig. 9, instead of treating the entire kernel domain as a whole, we divide it into a normal domain and a critical-object domain. Permissions of these critical objects are managed via an extra page and protection key. When entering helper functions, instead of setting the entire kernel space as access-enabled (AE), those critical objects remain access-disabled (AD), preventing the helpers from accessing them. To identify these objects, we first review BPF CVEs and find all objects that have been exploited. Then, we manually search for similar objects in the kernel and check that these found objects indeed contain sensitive fields (e.g., function pointers). It took two authors about half a month to conduct the above procedure individually and cross-check results, which ensures the credibility of our research to a great extent. We identified a total of 44 critical objects (see Appendix A); these objects could either leak the sensitive base address of the kernel (e.g., `iter_seq_info`) or even be tampered with to launch a full-blown exploit (e.g., `array_map_ops`). In addition to these 44 identified objects, we set MOAT itself and `cred` as critical objects. The former contains sensitive data of MOAT (e.g., the saved state of `IA32_PKRS`), while the latter tracks the privilege of a process. We believe protecting the identified critical objects provides a practical security guarantee for the BPF helpers. Nonetheless, unidentified critical objects could exist; we will discuss their potential threat in Sec. 8. Moreover, it is always feasible to extend COP to protect other kernel objects.

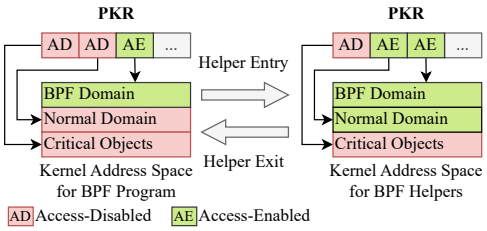

Figure 9: Critical object protection (COP).

4.3.2 Dynamic Parameter Auditing (DPA)

To further regulate the helpers, we propose Dynamic Parameter Auditing (DPA), which leverages the information obtained from the BPF verifier to dynamically check if the parameters are within their legitimate ranges. As illustrated in Fig. 10, the verifier can deduce the value range of each register via static analysis (aligned with the uncovered verifier inaccuracies [49, 53]; our DPA design tolerates even *invalidly* deduced value ranges; see clarification below). MOAT logs such value ranges and instruments the BPF programs to insert checks before helper calls. During runtime, these checks ensure that the provided parameters of the helpers are within the verifier-deduced value ranges during runtime. In our example, we can check if `r0==0x10; r1==0x11` when `BPF_HELPER` is called. If the parameter runtime values do not match with the static analysis results, the BPF program is terminated immediately.

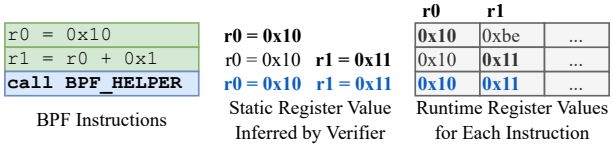

Figure 10: Register value tracking of the verifier.

Clarification. In this DPA strategy, one may wonder if the "value ranges" deduced by the verifier are wrong [49, 53]. To clarify this, we list possible cases of a BPF variable v's value range and the corresponding system states in Table 2; our discussions are as follows.

Table 2: Four cases of a BPF variable v's value ranges. R denotes the runtime value of v, D denotes the verifier's deduced value of v, E denotes verifier's *expected* legitimate value range of v, while T denotes the *ground truth* legitimate value range of v. The last column denotes this case is safe (✓), mitigated by verifier ($✓_V$), mitigated by MOAT ($✓_M$), or unsafe (✗)

	R	D	E	T	State
1	0x10	0x10	[0,0x20]	[0,0x20]	✓
2	**0xba**	0xba	[0,0x20]	[0,0x20]	$✓_V$
3	**0xba**	**0x10**	[0,0x20]	[0,0x20]	$✓_M$
4	**0xba**	**0xba**	**[0,0xba]**	[0,0x20]	✗

Case 1 illustrates the value range of a variable v in a benign BPF program. The runtime value aligns with verifier's deduction which further falls within the *expected* and *true* legitimate value ranges simultaneously ($R = D \in E = T$, see the caption of Table 2). Case 2 demonstrates the value range of variable v in a malformed BPF program. The runtime value `0xba` is out-of-bounds, and this invalid value is detected by the verifier through static analysis. Therefore, this program is rejected by the verifier, and the system remains safe ($R = D \notin E = T$). Case 3 shows the value range of v in a malicious BPF program. The runtime value `0xba` is out-of-bounds. However, due to the incomplete analysis caused by vulnerabilities, the verifier deduces that v's value is `0x10`, which is within the verifier's expectation. Since DPA operates in the runtime and checks whether the runtime value actually matches the verifier's deduction, this mismatch is then detected by DPA, and this malicious BPF program is terminated ($R \neq D \in E = T$).

While the above three cases cannot be exploited, Case 4 implies a scenario where DPA fails and the helper is abused. The verifier's *expected* value range differs from the *ground truth*, legitimate value range. This discrepancy allows an out-of-bounds value `0xba` to be passed as an argument to a helper for exploitation. For this to occur, the following conditions must be satisfied simultaneously: ① The verifier has an *incorrect* expectation (i.e., $E \neq T$). ② The incorrect expectation E is *unsafe* (i.e., $T - E$ overlaps an exploitable structure). ③ The BPF program is carefully tweaked to be aligned with D and evade DPA (i.e., $R = D$). For BPF programs, It is usually straightforward for the verifier to obtain E statically (e.g., E encodes the array size). It is thus hard to satisfy ① and ② simultaneously. For today's known BPF exploits (all of which fall into Case 3), the verifier has the correct expectation $E = T$ but makes the incomplete deduction $R \neq D$; therefore, the discrepancy $E \neq T$ is never encountered in practice.

4.4 Design Comparison

In this section, we compare MOAT's design with other works in kernel isolation. This helps highlight the contribution of MOAT, in comparison to previous research.

Virtualization. There is a line of research works on isolating kernel components via virtualization [16, 59, 60]. However, among these prior works, lightweight solutions like SKEE [16] are not compatible with the low-level routines (e.g., interrupt) in Linux. SKEE disables the interrupt upon entry, but disabling the interrupt will significantly impede BPF's network performance. To incorporate with low-level routines in the kernel, non-trivial modification to the system is often needed. For example, LVD and LXDs [59, 60] require a hypervisor and an implanted micro-kernel to manage the isolated components. MOAT, on the one hand, leverages PKS to enforce lightweight isolation between kernel and BPF; this ensures efficient interrupt handling. On the other hand, MOAT re-uses the kernel memory subsystem to enforce intra-BPF isolation without the additional hypervisor or micro-kernel.

SFI-based Solutions. Prior works also proposed Software Fault Injection (SFI) for kernel security [20, 41]. SFI inserts checks before memory access to ensure that they fall into valid ranges. However, inserting checks for memory access often brings higher overheads (see Sec. 6.3 for an empirical

comparison with MOAT). MOAT uses PKS to ensure memory safety and only inserts checks before helpers. Since the number of helper calls is much smaller than that of memory access, this design largely reduces the overhead of MOAT.

5 Implementation

MOAT is implemented on Linux 6.1.38, and consists of 2,911 lines of C code. We explain the key points below.

Kernel Interrupt Handling. MOAT has to cooperate with many low-level routines inside the kernel. For instance, during the execution of BPF programs, an interrupt may occur and take over the control flow to its handler. Note that most interrupt handlers require access to kernel memory, and as a result, the PKS would presumably raise spurious alerts. Thus, we need to temporarily disable PKS inside these handlers and re-enable it once the handlers finish. To avoid the overhead when there is no BPF program, we use a per-CPU variable `in_bpf` to identify whether the processor is executing a BPF program. Since BPF programs only occupy a tiny fraction of kernel execution time, we observe little performance loss due to this, even under cases where interrupt frequently occurs (e.g., intensive network activity in Sec. 6.2.2).

Granularity of PKS. As protection keys are associated with PTEs, MOAT only protects memory in the granularity of a page (i.e., 4 KB). However, the objects used by BPF programs may not be aligned to 4 KB, which means they could interleave with critical kernel structures. Therefore, granting BPF programs access to these objects also enables access to those kernel structures and leads to exploitation. To prevent this, we have modified BPF-related objects (e.g., maps) so that they are page-aligned and not interleaved with other structures.

DPA Check Generation. To deploy DPA from Sec. 4.3.2, we modify the BPF JIT compiler (`bpf_jit_comp.c` with about 2500 LoC) to instrument BPF programs. As shown in Fig. 11, our modified JIT compiler receives a set of expected ranges from the verifier. Then, for each parameter, the JIT compiler emits assembly instructions to check whether the parameter is within the expected range. If not, we terminate the program (`bad` label in Fig. 11). This prevents malicious programs from passing invalid parameters to abuse BPF helpers.

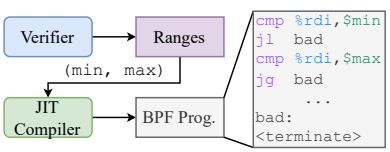

Figure 11: DPA Check Generation.

6 Evaluation

To evaluate MOAT, we first analyze how MOAT mitigates various attack interfaces and then benchmark its CVE detectability in Sec. 6.1. We then assess the performance of MOAT under different BPF program setups in Sec. 6.2.

6.1 Security Evaluation

6.1.1 Analysis of Attack Mitigation

We systematically analyze how MOAT mitigates the representative attacks on the BPF ecosystem as well as the potential threats to MOAT itself. Our analyses are illustrated in Fig. 12.

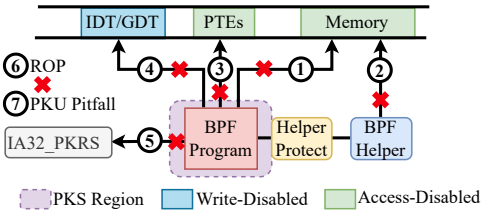

Figure 12: Analysis of attack mitigation.

① Arbitrary Kernel Access. Currently, the most prevalent threat to the BPF ecosystem is the ability of malicious BPF programs to arbitrarily modify kernel memory. In order to accomplish this, these BPF programs typically employ corner-case operations to deceive the verifier during the loading phase and to behave maliciously during runtime. This type of attack is effectively mitigated due to the fact that MOAT derives the necessary memory regions of each BPF program and uses PKS to prevent any runtime access beyond this region (Sec. 4.2), mitigating such illegal access.

② Helper Function Abuse. Apart from launching an attack directly from BPF programs, a malicious BPF program may carefully prepare parameter values and pass them to abuse certain helpers. To prevent such abuse, MOAT deploys security enforcement schemes (Sec. 4.3) to dynamically audit helper parameters and also protect critical kernel kernel objects during the execution of these helpers. Thus, the attacker can no longer take advantage of these helpers.

③ PTE Corruption. A page's PKS region is configured via its PTE. Consequently, a malicious BPF program may attempt to tamper with the PTEs to disable MOAT. However, this is impossible since MOAT sets these PTEs as access-disabled; they are thus protected by PKS like other kernel resources.

④ Descriptor Table Tampering. Descriptor tables like GDT and IDT are essential for segmentation and interrupt handling. Therefore, blindly setting them as access-disabled would cause system crashes. However, since these descriptor tables are only accessed in a read-only manner, MOAT sets them as write-disabled, thus preventing malicious BPF programs from using them to compromise the kernel.

⑤ Hardware Configuration Tampering. Besides memory-based attacks, attackers may also directly disable PKS through hardware configurations. As described in Sec. 2.2, `IA32_PKRS` is a critical register for configuring PKS. One may disable PKS by modifying `IA32_PKRS`. However, this register can only be modified via special instructions, and

BPF instruction sets do not include any of these. Thus, a BPF program with these instructions is rejected immediately. Since the BPF programs are set to $W \oplus X$ (meaning write and executable permissions are not simultaneously enabled), adding these instructions via self-modification is also impossible.

⑥ **Return-Oriented Programming.** Two properties of the BPF instruction set prevent potential control-flow hijacking attacks like return-oriented programming (ROP). First, BPF only supports jump instructions with *constant and instruction-level* offsets. This means the destinations of jumps are trivially known during the compile time, and there are no *unintended ROP gadgets* (jumps between instructions) like x86 [19]. Secondly, as a specialized instruction set, BPF does not include any instructions that may modify hardware configurations such as `XRSTOR` and `WRMSR`. These two properties allow MOAT to reliably detect invalid instructions and prevent BPF programs from tampering with hardware settings.

⑦ **Attacks in PKU Pitfalls.** We carefully examined attacks mentioned in PKU Pitfalls [25], which focus on breaking PKU, the user variant of MPK. Their noted attacks can be roughly categorized into three types. The first type manipulates memory mappings through certain system calls (e.g., `mremap`) to subvert PKU, such as modifying user-space PTEs and creating mutable backup. However, BPF programs are incapable of launching such attacks, as there is no helper that can manipulate kernel memory mappings. The second type involves tampering with the saved state of `PKRU` and disabling PKU entirely upon restoration. Unlike PKU, MOAT exclusively manages the saved state of `IA32_PKRS`, making these attacks infeasible. The third type relies on mechanisms that are exclusive to the user space (e.g., using `seccomp` to intercept system calls) and is not applicable to MOAT.

6.1.2 Real-world CVE Evaluation

We surveyed the BPF CVEs in the past ten years. A total of 26 CVEs are memory exploits (Appendix. B) and thus fall within the scope of MOAT. We tested MOAT's effectiveness on all of these CVEs. For CVEs with publicly available PoC, we ported and ran the PoC on MOAT-enabled kernel. For CVEs without PoC, we studied the fixes and ensure that MOAT mitigates them. In sum, we report that MOAT successfully mitigates *all* of them. We now present the following case studies.

CVE Case Study. To better explain how MOAT mitigates these CVEs, we elaborate on the exploit paths for three of them, 2022-23222, 2020-27194, and 2021-34866.

CVE-2022-23222 is a pointer mischeck vulnerability introduced via a rather new BPF feature, `bpf_ringbuf`. This new feature was brought to BPF in 2020, along with a new pointer type named `PTR_TO_MEM_OR_NULL`. However, the verifier had not been updated to track the bounds of this new type, resulting in this vulnerability. As shown in Fig. 13a, the malicious payload first retrieves a `nullptr` via `ringbuf_reserve` (line 1), which returns this newly added pointer type named `PTR_TO_MEM_OR_NULL`. Since this new type is not tracked by the verifier, the payload can bypass pointer checks by tricking the verifier that `r1` is `0x0` when it is `0x1` (line 3). `r1` can then be multiplied with any offset to perform arbitrary kernel access (line 6). However, such access violates PKS and is terminated by MOAT (line 7).

CVE-2020-27194 is a vulnerability due to incorrect truncation. As in Fig. 13b, the user first inputs an arbitrary value in the range of `[0,0x600000001]` (line 1). Then, the conditional clause helps the verifier to determine its value range (line 3). However, when tracking the `BPF_OR` operator, the verifier performs a wrong truncation on its upper bound. After the truncation, the user-controlled `r5` is viewed by the verifier as a legitimate constant `0x1` (line 5), which is later used as the offset to perform arbitrary access to the kernel (line 6). Similarly, such access is stopped by MOAT.

CVE-2021-34866 is a helper-abuse vulnerability. As shown in Fig. 13c, the malicious payload tries to pass an invalid map to the `ringbuf_reserve` to cause heap overflow (line 3). However, since the runtime value of `r5` does not match the argument of `ringbuf_reserve`, DPA prevents such a mismatched helper call (line 2). Moreover, supposing that the DPA is not enabled, and the helper tries to tamper with exploitable kernel objects (e.g., `array_map_ops`). COP protects these objects and thus prevents the helper from accessing them (line 3). Lastly, even if neither COP nor DPA is enabled, and the abused helper manages to return a leaked kernel pointer, accessing the leaked pointer violates PKS, and the malicious program is terminated by MOAT (line 4).

```
1 r0 = ringbuf_reserve(fd, INT_MAX, 0)
2 r1 = r0 + 1              // R:r0=0;r1=1 V:r0=r1=?
3 if (r0 != nullptr)       // R:r0=0;r1=1 V:r0=r1=?
4  exit(1)
5 off = <bad off>       // R:r0=0;r1=1 V:r0=r1=0
6 off = off * r1        // R:off=<bad off> V:off=0
7 *(ptr+off) = 0xbad    // PKS violation
```

(a) Code snippet of CVE-2022-23222

```
1 r5 = <bad addr>
2 r6 = 0x600000002
3 if (r5>=r6||r5<=0) // R&V:0x1<=r5<=0x600000001
4  exit(1)
5 r5 = r5 | 0       // R:r5=<bad addr> V: r5=0x1
6 *(ptr+r5)=0xbad   // PKS violation
```

(b) Code snippet of CVE-2020-27194

```
1 r5 = <bad map fd>
2 <DPA checks>                      // DPA violation
3 r0=ringbuf_reserve(r5,INT_MAX,0)// COP-guarded
4 *(r0+ptr_off) = 0xbad             // PKS violation
```

(c) Code snippet of CVE-2021-34866

Figure 13: CVE case study. `R` denotes variable runtime statuses. `V` denotes verifier-deduced values of variables.

6.2 Performance Evaluation

Evaluation Setup. We assess MOAT performance on Linux v6.1.38 and a 5-core Intel 8505 processor with PKS support. To reduce variance, hyper-threading, turbo-boost, and frequency scaling are disabled. All evaluated BPF programs are executed in the JIT mode, given that BPF JIT is enabled by default on all supported platforms. Moreover, both COP and DPA (Sec. 4.3) are enabled; COP is configured to protect the critical objects identified in Sec. 4.3.1. We manually inspected the CPU utilization to ensure it is close to 100%, and that the overhead is not hidden by the increased CPU load.

6.2.1 Micro Benchmark

For the micro benchmark, we measure the CPU cycles of four key operations in MOAT. We list the four operations in Table 3. `set_pkrs` changes region permissions by changing `IA32_PKRS` via `WRMSR`. `get_pkrs` returns the current permission configuration by reading `IA32_PKRS` via `RDMSR`. `bpf_{entry/exit}` is the total cost of entering/exiting a BPF program, which includes operations like swapping stack, managing BPF context, and configuring region permissions with `set_pkrs`. `dpa_check_args` is the cost of checking helper parameters. Each operation is measured by averaging ten runs of one million invocations to eliminate randomness. Since Intel has introduced the concept of "performance core" and "efficient core", we measure their cycles independently.

As shown in Table 3, the overall switching cost of MOAT is less than 200 cycles, which is negligible for most BPF programs (see Sec. 6.2.2 for details). Notably, setting and getting the region permissions (`set_pkrs`/`get_pkrs`) in PKS is more expensive than its user-space variant in `libmpk` [62] (see the caption of Table 3). We presume that this is because, in PKU, the region permission is controlled via a dedicated register named `PKRU` with two special instructions `RDPKRU`/`WRPKRU`, whereas in PKS employed by MOAT, its region permission is stored in an MSR named `IA32_PKRS` without any special instruction. To configure the permission in `IA32_PKRS`, one has to use the `RDMSR`/`WRMSR` instructions with the MSR ID `0x6E1`. Moreover, although the cost of `dpa_check_args` varies based on the checked range type (e.g., value point `[0x1, 0x1]` costs less than value range `[0x1, 0x10]`), we report that these costs are all less than ten cycles. Lastly, we observe that the operations of MOAT are not substantially affected by the difference between performance and efficient cores.

Table 3: Micro benchmark results. We use P to denote the cycles of performance cores and E for the efficient cores. As a reference [62], user-space `RDPKRU`, `WRPKRU` take 0.5 and 23.3 cycles, respectively.

Operation	#Cycles		Note
`get_pkrs/RDMSR`	P:36	E:43	Get region permissions
`set_pkrs/WRMSR`	P:111	E:112	Set region permissions
`bpf_{entry/exit}`	P:154	E:173	Entry/exit BPF program
`dpa_check_args`	P:≤10	E:≤10	Check helper arguments

MOAT's Overhead vs. #BPF Programs. To show MOAT supports that numerous BPF programs, we prepare the following experiments. We attach 1, 10, 32, 64, and 128 BPF programs to trace `execve`[3], run a program that continuously creates processes for one minute, and measure the number of processes created. In this setting, each invocation of `execve` stresses MOAT to constantly switch between the BPF programs. Moreover, we craft each BPF program as succinct (programs that directly return) so that MOAT's relative overhead is not "dominated" by the overly lengthy BPF programs.

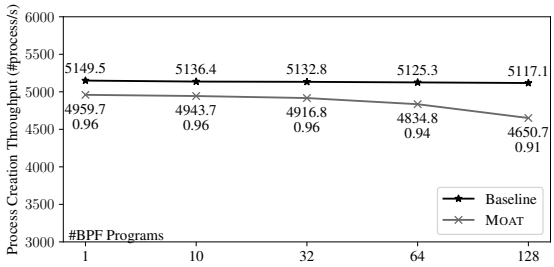

Figure 14: MOAT's overhead with respect to #BPF programs.

We report our results in Fig. 14. Since we use simple BPF programs, the baseline performance (without MOAT) is not observably affected by the number of BPF programs; we expect that in real-world cases, the baseline performance would also drop as the number of BPF programs increases. MOAT's overhead stays largely the same (4%) when there are 1, 10, or 32 BPF programs and then slowly increases with the number of BPF programs. It eventually incurs 9% overhead with 128 BPF programs. With further inspection, we find that over 64 BPF programs with isolated TLB entries pose heavy stress to TLB, resulting in increased overhead. However, having over 64 BPF programs attached to the same place is extremely rare (if it occurs at all). Nonetheless, even in such cases, MOAT incurs a performance penalty of less than 10%.

To complement the above experiment, we prepare the following experiment where BPF programs are attached to different kernel locations. There are, in total, 685 BPF tracepoints of system calls in Linux [15]. Following a similar setting as above, we attach each of these tracepoints with a simple BPF program and run UnixBench [42] to measure the overall system performance. In sum, there are nearly 700 BPF programs in the kernel with diverse invocation patterns. Thus, this setting stresses the system in a manner distinct from the previous experiment. We report that in such settings, the average UnixBench baseline score is 4936.3, MOAT's score is 4720.8, and thus, the incurred overhead is about 5%.

From these two experiments, we interpret that the overhead of MOAT is slightly affected by the number of BPF programs due to the TLB stress. Nevertheless, MOAT's overhead falls in a reasonable and promising range (4%~9%), even for cases

[3]We clarify that one BPF tracepoint only supports up to 64 programs [18], so we attach to both entry and exit tracepoints of `execve` in these experiments.

that are much heavier and rarely seen in real-world ones.

6.2.2 Macro Benchmark

For the macro benchmark, we set up three mainstream BPF use cases: network, system tracing, and system-call filtering. On the network cases (i.e., Socket and XDP), we use the smallest applicable packet size because BPF operates on each packet. Thus, under the same bandwidth, smaller-sized packets will incur higher throughput (i.e., more packets) and lead to more invocations of BPF programs, eventually putting more stress on MOAT.

Network — Socket. To evaluate MOAT's overhead on the network applications, we simulate a traffic monitoring scenario. A traffic generator sends UDP packets for one minute, with a packet size of 16 bytes, to our tested device. A server on the tested device receives these packets. Both the sender and receiver have a 1 GbE network interface controller. As for the tested BPF programs, we use five socket filtering programs from the Linux source tree, similar to previous works [39]:

- `drop`: directly ignores the packet.
- `byte`: monitors the traffic in bytes from each protocol.
- `pkt`: monitors the traffic in packets from each protocol.
- `trim`: only keeps the packet header to the socket.
- `flow`: monitors the network traffic based on protocol, interface, source, destination, and port.

We attach a socket to the receiver's network interface and set up these BPF programs to monitor the network traffic over the socket. In addition to the evaluation of each program, we also conduct a full-on experiment where five BPF programs are attached simultaneously to stress MOAT.

Table 4: MOAT's traffic monitoring performance in Thousand Packets per Second (TPPS). The full-on experiment is denoted as "all". The "vanilla" throughput without BPF program is 596.3 TPPS; the relative throughput is denoted in parentheses, e.g., (99.73%).

Throughput (TPPS)	drop	byte	pkt	trim	flow	all
Baseline	594.39 (99.70%)	594.67 (99.73%)	594.26 (99.66%)	594.74 (99.73%)	594.39 (99.68%)	587.22 (98.47%)
MOAT	593.10 (99.46%)	594.31 (99.66%)	594.43 (99.68%)	594.69 (99.73%)	593.10 (99.46%)	575.33 (96.48%)

As shown in Table 4, MOAT incurs negligible overhead (<1%) for all BPF programs if they are solely executing in the kernel. Even in the full-on experiment, which forces MOAT to constantly switch between these BPF programs, MOAT brings only a very small throughput drop of 2%.

Network — XDP. Besides processing packets from a socket buffer, BPF provides a direct way to control the network — eXpress Data Path (XDP). XDP processes packets at an early stage in the network stack to achieve fast packet processing. Following the settings in the socket experiment, we simulate a packet processing scenario. Similar to prior works [39], we run five XDP programs from the Linux source tree:

- `xdp1`: parses the IP header, keeps packets count in a BPF map, and drops the packets.
- `xdp2`: same as `xdp1`, but re-sends the packets.
- `adj`: trims the packets into ICMP packets, sends them back, and keeps packet count in a BPF map.
- `rxq1`: counts and drops the packets in each receive queue.
- `rxq2`: same as `rxq1`, but re-sends the packets.

Unlike socket filters, XDP programs require packets to be over a certain size, so we tune our traffic generator to send packets that go through the maximum possible execution path of the tested programs. We send packets of 64 bytes for `xdp1` and `xdp2` and packets of 100 bytes for `adj`, `rxq1`, and `rxq2`.

Table 5: MOAT's XDP performance in TPPS. The "vanilla" throughput without XDP program is 532.9 TPPS with 100-byte packets, and 561.5 TPPS with 64-byte packets; the relative throughput is denoted in parentheses, e.g., (99.55%).

Throughput (TPPS)	xdp1	xdp2	adj	rxq1	rxq2
Baseline	560.58 (99.84%)	557.78 (99.34%)	531.11 (99.66%)	528.36 (99.15%)	530.52 (99.55%)
MOAT	560.15 (99.76%)	557.76 (99.33%)	530.65 (99.58%)	527.57 (99.00%)	527.66 (99.05%)

As illustrated in Table 5, MOAT incurs negligible performance penalties (<1%) when executing XDP programs.

System Tracing. System tracing is another mainstream BPF use case. To evaluate MOAT's overhead on system tracing, we prepare 11 BPF programs to trace frequent system events like page faults, process creation, context switch, and file operations. These programs collect relevant system statistics for user-space analysis. Then, we run UnixBench [42] to measure the overall system performance. UnixBench includes the following tests: ① execl throughput, ② file copy, ③ pipe throughput, ④ pipe-based context switching, ⑤ process creation, ⑥ shell scripts, and ⑦ system call.

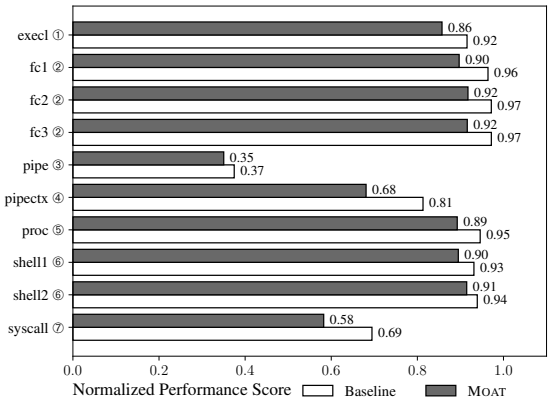

Figure 15: UnixBench normalized scores with respect to the "vanilla" scores without BPF tracepoints. The "fc* ②" and "shell* ⑥" are file copy tests and shell tests with different settings.

We report the results in Fig. 15. We find that MOAT imposes a small slowdown ($\leq 6\%$) for most UnixBench tests.

The maximum performance loss brought by MOAT is 13% in test ⑦. Such overhead seems moderate. However, note that the BPF programs without MOAT already bring a non-trivial performance penalty (e.g., 63% slowdown in test ③). Therefore, the performance loss brought by MOAT (<13%) is reasonably low for system tracing.

Syscall Filtering. BPF is also used to enhance software security [12, 28, 36]. `seccomp-BPF` allows filtering the system calls of a process with BPF. `sysfilter` [28] is an automated tool that analyzes a program, creates the set of system calls the program needs, and restricts the program using `seccomp-BPF`. We use `seccomp-BPF` and `sysfilter` to evaluate MOAT's overhead in such a use case. We apply the MOAT-hardened filter to Nginx and benchmark it using `wrk` [33] with one, two, and three client processes; each sends requests for one minute with 128 connections. Nginx is configured with the same number of worker processes. To clarify this setting: the number of processes for Nginx typically shall not exceed the number of cores, and adding more processes does not increase the throughput due to the context-switch cost. All requests are sent over the loopback (`lo`) to minimize network interference.

Table 6: Nginx throughput in Thousands of Request per Second (Treq/s). The relative throughput is in the parentheses, e.g., (95.6%).

Throughout (Treq/s)	1 worker	2 worker	3 worker
Vanilla (no `seccomp-BPF`)	148.1 (100%) ±12.81	179.5 (100%) ± 8.35	165.2 (100%) ±4.72
Baseline	147.2 (99.4%) ±9.56	171.3 (95.4%) ±8.08	160.5 (97.2%) ±5.28
MOAT	142.3 (96.1%) ±8.77	166.3 (92.6%) ±6.70	158.0 (95.6%) ±4.48

As shown in Table 6, MOAT incurs an additional throughput drop of 3%. Moreover, the standard deviations of throughput are within the normal range. Therefore, MOAT does not introduce fluctuation to Nginx throughput.

6.3 Additional Evaluation

In addition to the above experiments, we also evaluate MOAT's memory footprints, instrumentation cost from DPA, and compare MOAT's performance with a prior work [41].

Memory Footprint. As mentioned in Sec. 5, MOAT aligns BPF-related objects (e.g., maps) to 4 KB to ensure that they do not interleave with other kernel structures, introducing extra memory footprints. We provide a detailed breakdown of MOAT's memory footprints in Table 7. Specifically, MOAT uses four pages to set up the page table of isolated address-spaces, one page for the stack and one for the context. As for critical objects identified in Sec. 4.3.1, MOAT uses an extra page to toggle their permissions independently. Additionally, the memory used by the BPF program binaries and BPF maps is aligned up to a multiple of page size.

Though MOAT's memory footprints seem non-trivial from Table 7, we clarify that they are static and thus do not grow dynamically during the runtime. Even if there are thousands of BPF programs, MOAT's memory footprints are merely a few megabytes, which is negligible for modern systems.

Table 7: Breakdown of MOAT's memory footprint. AS: Address Space; ST: Stack; Ctx: Context; CO: Critical Objects. P and M denote #pages of program and map, respectively.

Type	AS	ST	CO	Ctx	Prog	Map
#Page	4	1	1	1	$\lceil P \rceil$	$\lceil M \rceil$

Helper Instrumentation Cost. As described in Sec. 4.3.2, MOAT instruments the BPF programs to insert DPA checks. Table 8 shows the number of helper calls and memory access made by the BPF programs[4] in Sec. 6.2.2. The former reflects the number of DPA checks inserted by MOAT, while the latter reflects the number of checks from SFI-based solutions. We report that, in most cases, the number of DPA checks is smaller than that of SFI-based solutions, let alone that SFI only offers memory safety, which is offered by PKS in MOAT.

Table 8: The number of checks of DPA and SFI-based isolation. Note that the tracepoint (marked with *) consists multiple BPF programs, and we report their average number of inserted checks.

Name	drop	byte	pkt	trim	flow	xdp1	xdp2	adj	rxq	tracepoint*
#Helper	0	1	1	0	2	3	3	5	3	6.4
#Mem.	0	4	3	1	61	16	29	53	37	6.9

Comparison with SandBPF. As far as we know, SandBPF [41] is the only work on BPF isolation at the time of writing. In this section, we compare MOAT with SandBPF. Technically, SandBPF enforces isolation by inserting software runtime checks into the memory access of BPF programs. As a result, it shall incur a relatively higher overhead than hardware-based methods. To substantiate this observation, we conduct the following direct comparative study. The authors of SandBPF conduct an evaluation with Phoronix Test Suite [40]; we reproduce the same experiments on MOAT. Note that the source code of SandBPF is not public at the time of writing, so we directly refer to their data in Table 9.

Table 9: Comparison with SandBPF [41]. We provide MOAT's relative overhead (Rel.) and SandBPF's overhead (Ref.).

Test #Conn (req./s)	XDP				Socket Filter			
	Base	MOAT	Rel.	Ref.	Base	MOAT	Rel.	Ref.
Apache 20	34,303	33,689	2%	**0%**	40,666	40,286	**1%**	4%
Apache 100	31,929	30,726	**4%**	8%	37,998	36,546	**4%**	4%
Apache 200	27,751	26,657	**4%**	5%	32,652	31,344	4%	**3%**
Apache 500	24,786	24,439	**1%**	7%	30,262	29,423	**3%**	7%
Apache 1000	24,597	24,470	**1%**	6%	29,545	28,961	**2%**	7%
Nginx 20	22,688	21,892	**3%**	7%	23,359	23,530	**0%**	10%
Nginx 100	21,492	20,689	**4%**	7%	22,870	22,482	**2%**	8%
Nginx 200	19,972	19,216	**4%**	6%	21,562	20,984	**3%**	8%
Nginx 500	18,470	17,814	**4%**	6%	19,421	18,713	**4%**	7%
Nginx 1000	17,024	16,735	**2%**	3%	17,392	17,098	**2%**	6%

As shown in Table 9, MOAT's overhead is lower than SandBPF in most testcases (Rel. v.s. Ref.). The MOAT's highest overhead is 4%, while SandBPF's is 10%. Again, we interpret the advantage of MOAT is attributed to the reduced

[4] The BPF programs in syscall filtering scenario are cBPF programs and thus do not support helper functions; we do not include them here.

number of inserted checks compared to SFI-based approaches.

7 Related Work

In this section, we discuss other related works on MPK and BPF. We already reviewed highly relevant works on kernel isolation and compared MOAT's design with theirs in Sec. 4.4.

MPK-based Isolation. Prior to PKS, Intel first announced its user-space variant PKU. Consequently, most existing works [35, 62, 70] using MPK focus on user-space isolation. To better utilize PKU as an isolation primitive, Park et al. [62] proposed `libmpk`, which resolves the semantic discrepancies between PKU and conventional `mprotect`. VDom and EPK [35, 70] aim to provide unlimited keys in the *user space* via key virtualization. Despite the similarity, we clarify that isolating BPF programs in the *kernel* is a distinct scenario and comes with its own challenges. This is why we have proposed the lightweight two-layer design to efficiently isolate BPF programs. Apart from using PKU to isolate user applications, efforts are made to isolate trusted applications in SGX via PKU [23, 45]. SGXLock [23] establishes mutual distrust between the kernel and the trusted SGX applications, while EnclaveDom [45] enables intra-isolation within one enclave. PKU has also been used for kernel security. IskiOS [34] applies PKU to kernel pages by marking them as user-owned.

BPF Security. There are prior works [32, 43, 44, 61, 69] on securing the BPF ecosystem. Most of them use formal methods to enhance the following BPF components: the verifier, the JIT compiler, or the BPF program. To enhance the BPF verifier, Gershuni et al. [32] propose PREVAIL based on abstract interpretation, which supports more program structures (e.g., loops) and is more efficient than the standard verifier. PRSafe [44], on the other hand, designs a new domain-specific language, whose ultimate goal is to build a mathematically verifiable compiler for BPF programs. Jitk [69] offers a JIT compiler whose correctness is proven manually, and Nelson et al. [61] generate automated proof for real-world BPF JIT compilers. Lastly, Luke Nelson [43] build proof-carrying BPF programs, requiring developers to provide a correctness proof with the program before loading it into the kernel.

8 Limitations

In this section, we discuss the limitations of MOAT, including the unidentified critical objects, issues brought by the granularity of PKS, and potential barriers to deploying MOAT.

Unidentified Critical Objects. In Sec. 4.3.1, we manually identified the critical objects in the BPF subsystem that have been exploited in the wild or similar to the ones that have been exploited. Despite two authors' rich experience, there might still exist unidentified critical objects. However, to exploit these unidentified objects, one still has to find a BPF helper that can be abused to access these critical objects and bypass the parameter checks (i.e., DPA) enforced by MOAT. Thus, even if few unidentified critical objects exist, they would be hard, if at all possible, to exploit due to DPA.

Page-size Granularity. MOAT leverages PKS to enforce isolation, which only supports 4 KB granularity. Though we carefully adjust BPF-related objects so that they are page-aligned, there still exist few corner-cases where PKS does not apply. In particular, BPF programs shall only access certain fields of the context `sk_buff`, which is enforced via the static checks by the verifier. However, applying PKS to these fields would significantly bloat `sk_buff` due to the granularity. MOAT thus cannot use PKS to constrain the access of BPF programs. As a result, `sk_buff` might have some bits leaked to BPF programs if the verifier's static checks are bypassed. Fortunately, BPF programs only receive a copy of `sk_buff` and cannot tamper with the original structure. Therefore, the consequence of such leakage, per our observation, seems trivial.

Barrier to Deploying MOAT. To deploy MOAT on other platforms or systems, there exist several barriers. First, MOAT requires a hardware feature like PKS that can provide page-level isolation. Fortunately, major platforms already support security features with similar capability. For example, on RISC-V, a prior work [64] implements a PKS-like feature named Donky, which could be used to support MOAT. Therefore, we expect MOAT to be deployable on other platforms with moderate engineering effort. Second, as we mentioned in Sec. 6.3, MOAT introduces additional memory footprints. Though the introduced footprints are only a few pages and negligible for modern systems, it might still create obstacles for some embedded systems, where memory is a scarce resource. Third, MOAT consists of about 3,000 lines of C code, which requires moderate engineering effort to port to other platforms.

9 Conclusion

Despite using BPF to extend kernel functionality, malicious BPF applications can bypass static security checks and conduct unauthorized kernel accesses. We present MOAT to isolate potentially malicious BPF applications from the kernel. MOAT delivers practical and extensible protection with a low cost, in compensation to contemporary BPF verifiers.

Acknowledgments

We are grateful to the anonymous reviewers, Dr. Zhou Lei, Dr. Adrian Rowland, and the members of COMPASS Lab for their valuable comments. This work is partly supported by the National Natural Science Foundation of China under Grant No.62372218 and Shenzhen Science and Technology Program under Grant No.SGDX20201103095408029. The HKUST authors are supported in part by a RGC CRF grant under the contract C6015-23G.

References

[1] Unprivileged bpf(), Oct 2015. URL https://lwn.net/Articles/660331/.

[2] cBPF is frozen without fix/extension, May 2019. URL https://lwn.net/ml/netdev/20190509044720.fxlcldi74atev5id@ast-mbp/.

[3] Reconsidering unprivileged BPF, Aug 2019. URL https://lwn.net/Articles/796328/.

[4] *BPF-Helpers(7) - Linux Manual Page*, 2021. URL https://man7.org/linux/man-pages/man7/bpf-helpers.7.html.

[5] eBPF seccomp() filters, May 2021. URL https://lwn.net/Articles/857228/.

[6] *Capabilities - Overview of Linux capabilities — The Linux manual page*, 2022. URL https://man7.org/linux/man-pages/man7/capabilities.7.html.

[7] *BPF Documentation — The Linux Kernel Documentation*, 2022. URL https://docs.kernel.org/bpf/index.html.

[8] *eBPF Maps — The Linux Kernel Documentation*, 2022. URL https://docs.kernel.org/bpf/maps.html.

[9] *eBPF Verifier — The Linux Kernel Documentation*, 2022. URL https://docs.kernel.org/bpf/verifier.html.

[10] *Kprobes Documentation — The Linux Kernel Documentation*, 2022. URL https://docs.kernel.org/trace/kprobes.html.

[11] *Intel 64 and IA-32 Architectures Software Developer Manuals*, 2022. URL https://www.intel.com/content/www/us/en/developer/articles/technical/intel-sdm.html.

[12] *Seccomp BPF (SECure COMPuting with filters)*, 2022. URL https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html.

[13] Unprivileged BPF and authoritative security hooks, Apr 2023. URL https://lwn.net/Articles/929746/.

[14] MOAT website, Oct 2023. URL https://sites.google.com/view/safe-bpf/.

[15] Analysing behaviour using events and tracepoints, 2023. URL https://www.kernel.org/doc/Documentation/trace/tracepoint-analysis.txt.

[16] Ahmed M. Azab, Kirk Swidowski, Rohan Bhutkar, Jia Ma, Wenbo Shen, Ruowen Wang, and Peng Ning. Skee: A lightweight secure kernel-level execution environment for arm. In *Network and Distributed System Security Symposium*, 2016. URL https://api.semanticscholar.org/CorpusID:8991310.

[17] Bootlin. BPF Configuration Map — config_map. https://elixir.bootlin.com/linux/v5.10/source/samples/bpf/xdp_rxq_info_kern.c#L32, 2023.

[18] Bootlin. Linux — BPF_TRACE_MAX_PROGS. https://elixir.bootlin.com/linux/latest/source/kernel/trace/bpf_trace.c#L2115, 2023.

[19] Nicholas Carlini and David Wagner. ROP is still dangerous: Breaking modern defenses. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 385–399, USA, 2014. USENIX Association.

[20] Miguel Castro, Manuel Costa, Jean-Philippe Martin, Marcus Peinado, Periklis Akritidis, Austin Donnelly, Paul Barham, and Richard Black. Fast byte-granularity software fault isolation. In *Proceedings of the 22nd ACM Symposium on Operating Systems Principles 2009, SOSP 2009, Big Sky, Montana, USA, October 11-14, 2009*, pages 45–58. ACM, 2009. doi: 10.1145/1629575.1629581.

[21] Sunjay Cauligi, Craig Disselkoen, Daniel Moghimi, Gilles Barthe, and Deian Stefan. Sok: Practical foundations for software spectre defenses. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 666–680, 2022. doi: 10.1109/SP46214.2022.9833707.

[22] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten-Hwang Lai. SgxPectre attacks: Leaking enclave secrets via speculative execution. *CoRR*, abs/1802.09085, 2018.

[23] Yuan Chen, Jiaqi Li, Guorui Xu, Yajin Zhou, Zhi Wang, Cong Wang, and Kui Ren. SGXLock: Towards efficiently establishing mutual distrust between host application and enclave for SGX. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 4129–4146, Boston, MA, August 2022. USENIX Association.

[24] Cilium. Cilium. https://github.com/cilium/cilium, 2022.

[25] R. Joseph Connor, Tyler McDaniel, Jared M. Smith, and Max Schuchard. PKU pitfalls: Attacks on pku-based memory isolation systems. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1409–1426. USENIX Association, 2020.

[26] Jinhua Cui, Jason Zhijingcheng Yu, Shweta Shinde, Prateek Saxena, and Zhiping Cai. SmashEx: Smashing SGX enclaves using exceptions. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 779–793. ACM, 2021. doi: 10.1145/3460120.3484821.

[27] CVEDetails. CVE-2020-27171. https://www.cvedetails.com/cve/CVE-2020-27171.

[28] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. sysfilter: Automated system call filtering for commodity software. In *23rd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2020)*, pages 459–474, San Sebastian, October 2020. USENIX Association.

[29] Henri Maxime Demoulin, Isaac Pedisich, Nikos Vasilakis, Vincent Liu, Boon Thau Loo, and Linh Thi Xuan Phan. Detecting asymmetric application-layer Denial-of-Service attacks In-Flight with FineLame. In *Proceedings of the 2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 693–708, Renton, WA, 2019. USENIX Association.

[30] Pekka Enberg, Ashwin Rao, and Sasu Tarkoma. Partition-aware packet steering using XDP and eBPF for improving application-level parallelism. In *Proceedings of the 1st ACM CoNEXT Workshop on Emerging in-Network Computing Paradigms, ENCP@CoNEXT 2019, Orlando, FL, USA, December 9, 2019*, pages 27–33. ACM, 2019. doi: 10.1145/3359993.3366766.

[31] Facebook. Katran: A high performance layer 4 load balancer, Oct 2023. URL https://github.com/facebookincubator/katran/tree/main.

[32] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A. Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1069–1084, New York, NY, USA, 2019. Association for Computing Machinery. doi: 10.1145/3314221.3314590.

[33] Will Glozer. wrk - a http benchmarking tool, Feb 2021. URL https://github.com/wg/wrk.

[34] Spyridoula Gravani, Mohammad Hedayati, John Criswell, and Michael L. Scott. Fast intra-kernel isolation and security with IskiOS. In *24th International Symposium on Research in Attacks, Intrusions and Defenses*, RAID '21, page 119–134, New York, NY, USA, 2021. Association for Computing Machinery. doi: 10.1145/3471621.3471849.

[35] Jinyu Gu, Hao Li, Wentai Li, Yubin Xia, and Haibo Chen. EPK: Scalable and efficient memory protection keys. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, pages 609–624, Carlsbad, CA, July 2022. USENIX Association.

[36] Wanning He, Hongyi Lu, Fengwei Zhang, and Shuai Wang. Ringguard: Guard io_uring with eBPF. In *Proceedings of the 1st Workshop on eBPF and Kernel Extensions*, eBPF '23, page 56–62, New York, NY, USA, 2023. Association for Computing Machinery.

[37] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. The EXpress data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies*, CoNEXT '18, page 54–66, New York, NY, USA, 2018. Association for Computing Machinery. doi: 10.1145/3281411.3281443.

[38] Canonical Inc. Unprivileged ebpf disabled by default for ubuntu 20.04 lts, 18.04 lts, 16.04 esm, Mar 2022. URL https://discourse.ubuntu.com/t/unprivileged-ebpf-disabled-by-default-for-ubuntu-20-04-lts-18-04-lts-16-04-esm/27047.

[39] Di Jin, Vaggelis Atlidakis, and Vasileios P. Kemerlis. EPF: Evil packet filter. In *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pages 735–751, Boston, MA, USA, 2023. USENIX Association.

[40] Michael Larabel. Phoronix test suite, Mar 2024. URL https://www.phoronix-test-suite.com/.

[41] Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. Unleashing unprivileged ebpf potential with dynamic sandboxing. In *Proceedings of the 1st Workshop on EBPF and Kernel Extensions*, eBPF '23, page 42–48, New York, NY, USA, 2023. Association for Computing Machinery. doi: 10.1145/3609021.3609301.

[42] Kelly Lucas. UnixBench: the original BYTE UNIX benchmark suite, updated and revised by many people over the years., Apr 2023. URL https://github.com/kdlucas/byte-unixbench.

[43] Emina Torlak Luke Nelson, Xi Wang. A proof-carrying approach to building correct and flexible in-kernel verifiers. Linux Plumbers Conference, 2021.

[44] Sai Veerya Mahadevan, Yuuki Takano, and Atsuko Miyaji. PRSafe: Primitive recursive function based

domain specific language using llvm. In *2021 International Conference on Electronics, Information, and Communication (ICEIC)*, pages 1–4, 2021. doi: 10.1109/ICEIC51217.2021.9369763.

[45] Marcela S. Melara, Michael J. Freedman, and Mic Bowman. EnclaveDom: Privilege separation for large-tcb applications in trusted execution environments, 2019.

[46] Dirk Merkel. Docker: lightweight linux containers for consistent development and deployment. *Linux journal*, 2014(239):2, 2014.

[47] MITRE. CVE-2020-27194. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-27194, .

[48] MITRE. CVE-2020-8835. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2020-8835, .

[49] MITRE. CVE-2021-31440. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-31440, .

[50] MITRE. CVE-2021-33200. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-33200, .

[51] MITRE. CVE-2021-3444. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3444, .

[52] MITRE. CVE-2021-34866. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-34866, .

[53] MITRE. CVE-2021-3490. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3490, .

[54] MITRE. CVE-2021-45402. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-45402, .

[55] MITRE. CVE-2022-23222. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-23222, .

[56] MITRE. CVE-2022-2785. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-CVE-2022-2785, .

[57] MITRE. CVE-2021-4001. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-4001, .

[58] MITRE. CVE-2021-29155. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-29155, .

[59] Vikram Narayanan, Abhiram Balasubramanian, Charlie Jacobsen, Sarah Spall, Scott Bauer, Michael Quigley, Aftab Hussain, Abdullah Younis, Junjie Shen, Moinak Bhattacharyya, and Anton Burtsev. LXDs: Towards isolation of kernel subsystems. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 269–284, Renton, WA, July 2019. USENIX Association. URL https://www.usenix.org/conference/atc19/presentation/narayanan.

[60] Vikram Narayanan, Yongzhe Huang, Gang Tan, Trent Jaeger, and Anton Burtsev. Lightweight kernel isolation with virtualization and vm functions. In *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, VEE '20, page 157–171, New York, NY, USA, 2020. Association for Computing Machinery. doi: 10.1145/3381052.3381328.

[61] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the linux kernel. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 41–61. USENIX Association, November 2020.

[62] Soyeon Park, Sangho Lee, Wen Xu, HyunGon Moon, and Taesoo Kim. libmpk: Software abstraction for intel memory protection keys (Intel MPK). In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 241–254, Renton, WA, July 2019. USENIX Association.

[63] IO Visor Project. BPF Compiler Collection. https://github.com/iovisor/bcc, 2022.

[64] David Schrammel, Samuel Weiser, Stefan Steinegger, Martin Schwarzl, Michael Schwarz, Stefan Mangard, and Daniel Gruss. Donky: Domain keys – efficient In-Process isolation for RISC-V and x86. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1677–1694. USENIX Association, August 2020.

[65] SUSE Support. Security hardening: Use of ebpf by unprivileged users has been disabled by default, Jan 2022. URL https://www.suse.com/support/kb/doc/?id=000020545.

[66] tr3e. CVE-2022-23222: Linux Kernel eBPF Local Privilege Escalation, Jun 2022. URL https://github.com/tr3ee/CVE-2022-23222.

[67] Anjo Vahldiek-Oberwagner, Eslam Elnikety, Nuno O. Duarte, Michael Sammler, Peter Druschel, and Deepak Garg. ERIM: Secure, efficient in-process isolation with protection keys MPK. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1221–1238, Santa Clara, CA, August 2019. USENIX Association.

[68] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. Sound, precise, and fast abstract interpretation with tristate numbers. In *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '22, page 254–265. IEEE Press, 2022. doi: 10.1109/CGO53902.2022.9741267.

[69] Xi Wang, David Lazar, Nickolai Zeldovich, Adam Chlipala, and Zachary Tatlock. Jitk: A trustworthy In-Kernel

interpreter infrastructure. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 33–47, Broomfield, CO, October 2014. USENIX Association.

[70] Ziqi Yuan, Siyu Hong, Rui Chang, Yajin Zhou, Wenbo Shen, and Kui Ren. VDom: Fast and unlimited virtual domains on multiple architectures. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ASPLOS 2023, page 905–919, New York, NY, USA, 2023. Association for Computing Machinery. doi: 10.1145/3575693.3575735.

[71] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, and Asaf Cidon. XRP: In-Kernel storage functions with eBPF. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 375–393, Carlsbad, CA, July 2022. USENIX Association.

[72] Hao Zhou, Shuohan Wu, Xiapu Luo, Ting Wang, Yajin Zhou, Chao Zhang, and Haipeng Cai. NCScope: hardware-assisted analyzer for native code in android apps. In Sukyoung Ryu and Yannis Smaragdakis, editors, *ISSTA 22: 31st ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, South Korea, July 18 - 22, 2022*, pages 629–641. ACM, 2022. doi: 10.1145/3533767.3534410.

Appendix A: Critical Objects

We list the critical objects identified by us in Table 10. We categorize the objects in Table 10 based on their locations. Despite different location (i.e., map and iterator), both of them are used to implement dynamic dispatching in the kernel.

Table 10: Critical objects in the BPF subsystem.

Location	Critical Object
Map	array_map_ops, percpu_array_map_ops, prog_array_map_ops, sock_map_ops, cgroup_storage_map_ops, htab_map_ops, htab_percpu_map_ops, htab_lru_map_ops, htab_lru_percpu_map_ops, trie_map_ops, task_storage_map_ops, dev_map_ops, sk_storage_map_ops, cpu_map_ops, xsk_map_ops, perf_event_array_map_ops, queue_map_ops, stack_map_ops, bpf_struct_ops_map_ops, ringbuf_map_ops, bloom_filter_map_ops,cgroup_storage_map_ops, cgroup_array_map_ops,array_of_maps_map_ops, stack_trace_map_ops, htab_of_maps_map_ops, user_ringbuf_map_ops, inode_storage_map_ops
Iterator	cgroup_iter_seq_info, sock_map_iter_seq_info, sock_hash_iter_seq_info, ksym_iter_seq_info, bpf_link_seq_info, bpf_map_seq_info, sock_hash_iter_seq_info, sock_map_iter_seq_info, ipv6_route_seq_info, iter_seq_info, ksym_iter_seq_info, netlink_seq_info, tcp_seq_info, udp_seq_info, unix_seq_info, bpf_prog_seq_info

Appendix B: BPF CVE List

We provide the list of evaluated BPF CVEs in Table 11.

Table 11: Evaluated BPF CVEs.

CVE ID
2016-2383, 2017-16995, 2017-16996, 2017-17852, 2017-17853, 2017-17854, 2017-17855, 2017-17856, 2017-17857, 2017-17862, 2017-17863, 2017-17864, 2018-18445, 2020-8835, 2020-27194, 2021-34866, 2021-3489, 2021-3490, 2021-20268, 2021-3444,2021-33200, 2021-45402, 2022-2785, 2022-23222, 2023-39191, 2023-2163

Appendix C: Supported BPF Helpers

We list the helper functions that are tested on MOAT in Table 12. To test these helper functions, we adapt the BPF programs included in the Linux kernel tree to invoke these helpers. We also check their results to ensure these helpers are executing correctly on a MOAT-enabled system.

Table 12: MOAT-supported helpers.

Type	Supported BPF Helpers
Map	bpf_map_lookup_elem, bpf_map_update_elem, bpf_map_delete_elem, bpf_map_push_elem, bpf_map_pop_elem, bpf_map_peek_elem
String	bpf_strtol, bpf_strtoul, bpf_strncmp
Utilities	bpf_trace_vprintk, bpf_get_retval, bpf_set_retval, bpf_user_rnd_u32, bpf_get_raw_cpu_id, bpf_get_smp_processor_id, bpf_ktime_get_ns, bpf_ktime_get_boot_ns, bpf_ktime_get_coarse_ns, bpf_get_current_pid_tgid, bpf_get_current_uid_gid, bpf_jiffies64, bpf_get_attach_cookie
Cgroup	bpf_get_current_cgroup_id, bpf_get_cgroup_classid_curr, bpf_get_cgroup_classid, bpf_skb_cgroup_id, bpf_sk_ancestor_cgroup_id, bpf_skb_cgroup_classid, bpf_sk_cgroup_id, bpf_skb_ancestor_cgroup_id, bpf_get_current_ancestor_cgroup_id
Tracing	bpf_probe_read_compat_str, bpf_probe_read_compat, bpf_probe_read_kernel_str, bpf_probe_read_kernel, bpf_get_current_task, bpf_get_func_ip_tracing, bpf_task_pt_regs, bpf_perf_event_read, bpf_perf_event_read_value, bpf_perf_event_output, bpf_get_func_ret, bpf_get_func_arg, bpf_get_func_arg_cnt, bpf_get_func_ip, bpf_get_ns_current_pid_tgid
Ringbuf	bpf_ringbuf_discard, bpf_ringbuf_query, bpf_ringbuf_submit, bpf_ringbuf_reserve, bpf_ringbuf_output
XDP	bpf_xdp_fib_lookup, bpf_xdp_load_bytes, bpf_xdp_store_bytes, bpf_xdp_adjust_head, bpf_xdp_adjust_meta, bpf_xdp_adjust_tail, bpf_xdp_get_buff_len
Socket	bpf_get_listener_sock, bpf_skb_get_pay_offset, bpf_skc_to_mptcp_sock, bpf_skc_to_tcp6_sock, bpf_skc_to_tcp_request_sock, bpf_skc_to_tcp_sock, bpf_skc_to_tcp_timewait_sock, bpf_skc_to_udp6_sock, bpf_skc_to_unix_sock, bpf_sk_fullsock, bpf_sk_release, bpf_tcp_sock, bpf_skb_load_helper_8_no_cache, bpf_skb_load_helper_16_no_cache, bpf_skb_load_helper_32_no_cache, bpf_sock_ops_cb_flags_set, bpf_task_storage_delete, bpf_skb_load_bytes, bpf_skb_load_helper_16, bpf_skb_load_helper_32, bpf_skb_load_helper_8