# HIVE: A Hardware-assisted Isolated Execution Environment for eBPF on AArch64

Peihua Zhang[1,2], Chenggang Wu[1,2,3], Xiangyu Meng [4†], Yinqian Zhang[5], Mingfan Peng[1,2], Shiyang Zhang[1,2], Bing Hu[1,2], Mengyao Xie[1], Yuanming Lai[1,2], Yan Kang[1,2], and Zhe Wang [1,2,3*]

[1]SKLP, Institute of Computing Technology, CAS
[2]University of Chinese Academy of Sciences
[3]Zhongguancun Laboratory
[4]Northwestern Polytechnical University
[5]Southern University of Science and Technology

## Abstract

eBPF has become a critical component in Linux. To ensure kernel security, BPF programs are statically verified before being loaded and executed in the kernel. However, the state-of-the-art eBPF verifier has both security and complexity issues. To this end, we choose to look at BPF programs from a new perspective and regard them as a new type of kernel-mode application, thus an isolation-based rather than a verification-based approach is needed. In this paper, we propose HIVE, an isolation execution environment for BPF programs on AArch64. To provide the equivalent security guarantees, we systematize the security aims of the eBPF verifier and categorize two types of pointers in eBPF: the *inclusive type* pointer that points to BPF objects and the *exclusive type* pointer that points to kernel objects. For the former, HIVE compartmentalizes all BPF memory from the kernel and *de-privileges* the memory accesses in the BPF programs by leveraging the *load/store unprivileged* instructions; for the latter, HIVE utilizes the *pointer authentication* feature to enforce access controls of kernel objects. Evaluation results show that HIVE is not only efficient but also supports complex BPF programs.

## 1 Introduction

Linux eBPF allows users to load programs into the kernel and customize its behavior without modifying the kernel code [11]. eBPF provides an instruction set and execution environment in the kernel. At load time, the BPF program undergoes a static verification to ensure kernel security. Then, it is compiled and mounted at a specified kernel location. BPF programs are allowed to call helper functions, which are offered by the kernel to enable interaction with the system.

While eBPF has been used in various scenarios [2, 4, 9, 13, 15, 15, 32, 34, 37], it presents two issues in use. The first is the *complexity issue*, where legal programs may fail in the verification due to the verifier's limited capabilities [2, 13, 15, 37]. Researchers have resorted to "verifier-oriented programming" to circumvent this issue, such as masking memory accesses to reduce the complexity of verification [37]. Even so, it remains a persistent issue highlighted by many literature [13, 15, 37]; The second is the *security issue*, where malicious programs may pass the verification due to vulnerabilities [3, 14, 18, 31]. According to statistics, over half (36/60) of eBPF's CVEs come from the verifier since 2014 [31].

Through systematic analysis, we found that the above issues primarily come from the *full path analysis* stage of the verification (§3). It executes symbolically the program at the entry and explores all possible execution paths to check whether the state is illegal or not. However, it encounters the well-known *state explosion* problem, which significantly limits the accuracy of such analysis. Therefore, the verification-based method has become the bottleneck of eBPF.

In this work, we aim to address the above challenges in eBPF, to enable its broader practical application. Specifically, current BPF programs are considered part of the (untrusted) kernel code, so eBPF uses the verification-based method to "review" the code to identify all abnormal behaviors. But we choose another perspective — *BPF programs are no longer part of the kernel code, but a new type of kernel-mode application that interacts with the kernel through helper function calls rather than system calls, so kernel security should be achieved by isolating BPF programs, not by verification.*

As such, we propose to build an isolated execution environment for eBPF, dubbed HIVE[1], and enforce runtime isolation for BPF programs, thus eliminating the need for full-path analysis in the verification. Our design is based on AArch64, which is particularly motivated by the growing popularity of Arm-based processors in the mobile, PC, and server market, but should be also applicable to other ISAs, such as x86 (§9).

Our first step is to understand the security goals of the verifier in its full-path analysis. To this end, we conducted a comprehensive analysis of the verifier (§3) and concluded three security goals: 1) *memory safety* that ensures BPF pro-

†This work was done while he was a visiting student in ICT.

*Zhe Wang is the corresponding author. wangzhe12@ict.ac.cn.

[1] We call it HIVE to refer to a beehive for the bees in the eBPF logo.

grams cannot arbitrarily access kernel memory; 2) *leakage prevention* that prevents BPF programs from leaking kernel information; and 3) *DoS prevention* that ensures BPF programs cannot cause crashes or excessively consume CPU time.

The core approach taken by HIVE is to *de-privilege the BPF program*: the memory of the BPF program is mapped to an independent address space (called BPF space) and set to be un-privileged pages; the BPF program still runs at the privileged level, but is compiled to use the load/store un-privileged (LSU) instructions. The verifier, even without full-path analysis, can prevent illegal instructions and illegal control flows at load time. This simple setup naturally achieves the above goals: *Firstly*, as the kernel memory is set to be privileged pages, the BPF program cannot access them, thus ensuring memory safety. *Secondly*, the layout in the BPF space cannot be used to infer the kernel address space layout, thus preventing information leakage. *Thirdly*, HIVE captures exceptions and monitors the BPF program's execution time, rolls back the execution state, and uninstalls the BPF program once an exception or timeout is detected, thus preventing DoS.

However, it is difficult for BPF programs to be completely *decoupled* from the kernel, thus such a strict isolation execution environment is not practical. The *first challenge* is that there is a large amount of data in BPF objects that are inaccessible to BPF programs, such as metadata in maps. Simply mapping all memory into BPF space will face the problem of sub-page attacks; the *second challenge* is that legal access to kernel objects by de-referencing kernel pointers is also prohibited due to all instructions being emitted as LSU instructions; the *third challenge* is that it is difficult to prevent kernel pointers from being leaked without the ability to accurately identify them. We analyze all eBPF pointers, classify them into two categories, and process them separately in HIVE.

The *inclusive type pointers* can be de-referenced by the same memory access instruction, they point to BPF data objects and share the same logic of security checks that the pointer can only access the raw data inside its corresponding object, and it cannot be leaked. To ensure their security, HIVE compartmentalizes all BPF memory (e.g., stack and maps) and isolates all BPF-accessible memory regions to the BPF space. The pointers can be leaked safely due to the design of independent/decoupled address space.

The *exclusive type pointers* point to kernel objects, and a memory access instruction can only be used to de-reference a pointer of the same kernel object type. However, the de-referenced points are unknown without the full-path analysis. To address this, HIVE proposes the *exception-based point-of-use probing* method to dynamically identify the de-referenced points, perform security checks, and patch the code to use the regular load/store instruction to access the kernel object. To enforce memory access controls, HIVE utilizes the pointer authentication (PA) feature to ensure pointer integrity and type safety. Furthermore, HIVE also proposes *a descriptor-based method* to hide the real pointers.

We implemented and evaluated HIVE on Apple mini with M1 chips. The core of HIVE is a kernel module that manages the BPF space and monitors the exception. It also modifies the eBPF subsystem, such as removing the full-path analysis. We use two web servers (i.e., Nginx and Apache) and two databases (i.e., Memcached and Redis) to evaluate the performance of HIVE when running 161 BPF programs from BCC [35] and Tracee [4]. The experimental results show that HIVE is efficient compared with the vanilla eBPF. To evaluate the complexity promotion effectiveness, we successfully replaced 10 in-tree kernel modules with HIVE-equipped eBPF and applied HIVE to accelerate system calls.

To summarize, the contributions are as follows:

- **A comprehensive study on eBPF's verifier.** We conduct a comprehensive study on the verifier for the first time, resulting in a summary of its security properties and a discussion of its issues in complexity and security.
- **A novel isolated execution environment for BPF programs.** We propose a new isolated execution environment on AArch64, HIVE, to isolate the BPF programs by combining a set of hardware features, which could provide the same level of security guarantees as static verification.
- **New insights from implementation and evaluation**. We implement and evaluate a prototype of HIVE. The results show that the dynamic isolation for BPF programs can be practical, and it could replace the verification.

## 2 Background

### 2.1 Extended Berkeley Packet Filter (eBPF)

At load time, the eBPF verifier performs static analysis to ensure the kernel's security. Once the verification is passed, the program will be compiled into the native code and mounted to the specified kernel location. eBPF provides a RISC instruction set and an execution environment inside the kernel which offers the following supports.

- **Registers**. There are 11 registers (`r0-r10`) and will be mapped into physical registers after JIT compilation.
- **Maps.** Users can statically apply for custom-sized data areas through *maps*, which are key-value stores. Maps can be accessed by both user processes (via syscall) and BPF programs, enabling the data exchange between them.
- **Stack.** The BPF function's frame is stored on the BPF stack, which uses the kernel stack. BPF and kernel stack frames are arranged in the order of function calls.
- **Context.** When the kernel calls the BPF program, it passes a *context* structure, which is a program-type related kernel object, through the function parameter.
- **Helper functions.** Programs are allowed to call *helper functions*, which are dedicated functions offered by the kernel to enable interaction with the system.

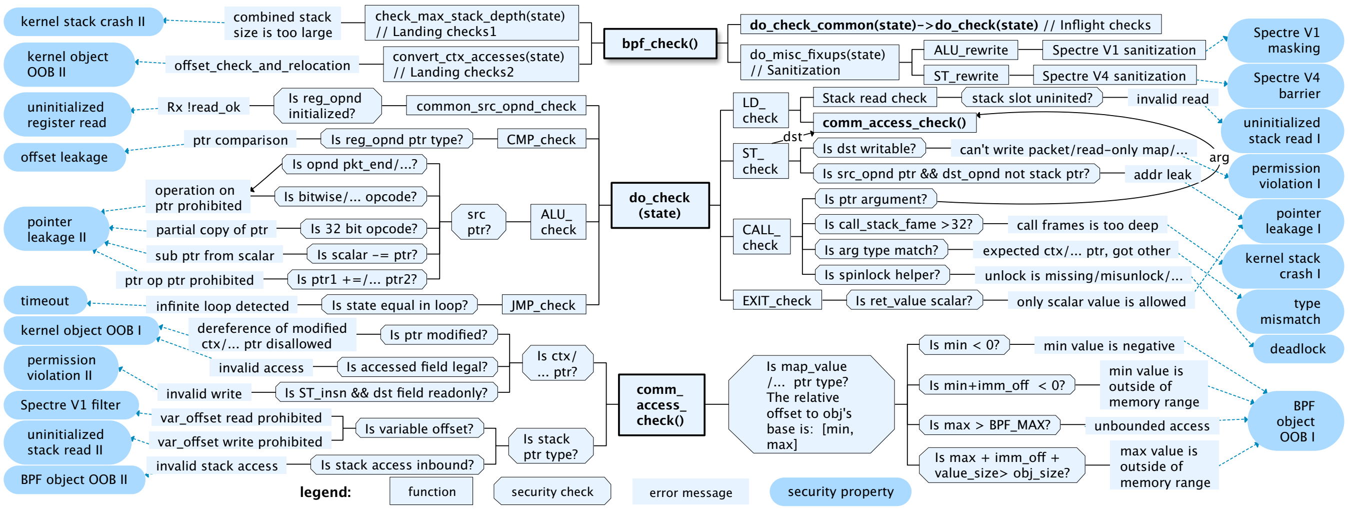


Fig. 1: The security checks and corresponding security properties of the verifier in the full-path analysis.

## 2.2 Hardware Features on AArch64

**Unprivileged load/store.** The access right of a memory access instruction is determined by the current exception level (EL) and the target memory's permission. The code runs at EL0 (i.e., the user mode) can access the unprivileged pages (abbreviated to *U-page*) in user space but cannot access the privileged page (abbreviated to *P-page*) in kernel space, and we call that *EL-based memory isolation*. But the *load/store unprivileged (LSU) instruction*s, i.e., `ldtr` and `sttr`, is an exception. No matter which EL they are executed at, they are treated as if at EL0, and thus cannot access P-pages.

**E0PD**. E0PD [6] is introduced in ARMv8.5-A as hardware mitigation to prevent the fault timing attacks launched by malicious users against the kernel. It prevents the unprivileged memory accesses to the (lower or upper or both) halves of the address space and generates the translation fault in constant time when accessing. There are two bits `E0PD0` and `E0PD1` of the `TCR_EL1` register which controls whether unprivileged memory accesses to the lower-half (user) or the upper-half (kernel) of the address space is disabled, respectively.

**Pointer Authentication**. Pointer Authentication (PA) [6] is a hardware feature in ARMv8.3-A that employs the cryptographic authentication code to protect the pointer's integrity. It attaches a PA Code (PAC) to a pointer in the unused bits of a 64-bit address. PA introduces `pac*` and `aut*` instruction families to *sign* and *authenticate* a pointer with a 64-bit *modifier* using the *key* indicated by the instruction name. [2]. The `pac*` instructions calculate the PAC and attach it to the pointer. The `aut*` instructions authenticate a pointer and clear the PAC in the pointer, making the pointer usable. If the authentication fails, the pointer will be modified to an invalid pointer.

# 3 Understanding the eBPF Verifier

## 3.1 The Workflow of the Verifier

The workflow of the verifier consists of three consecutive stages, with the later stages becoming more complex. The *pre-process* stage scans the BPF program linearly to find relocation items and check if there are any unallowed opcodes, e.g., indirect calls; The *CFG check* stage searches the control flow graph and forbids any out-of-bounds jumps or unreachable codes; The last stage creates a state machine that records the type and range of all registers and stack slots during the path exploration. Meanwhile, it verifies that all states conform to a set of security properties that do not threaten the kernel. Since it traverses all possible paths, we call it *full-path analysis* in this paper. The full-path analysis performs most of the analysis and integrates the majority of security checks, focusing on it allows for a comprehensive understanding of the verifier's capabilities and limitations.

## 3.2 The Internals of Full-path Analysis

### 3.2.1 The Methodology of Study

Due to the lack of comprehensive documentation explicitly about the verifier, we had to understand its design manually. To this end, we dedicated 80 person-days to meticulously review and understand the verifier's source code. We undertook a top-down analysis from the verifier's entry to all exits. We collected all checks and scrutinized the triggering conditions, the check's content, its dependencies, and restrictions.

All checks share a distinguishing feature, which is a conditional judgment statement with an error message output (e.g., `verbose()`). Among these checks, some of them are not related to the full-path analysis. For example, the division-

[2] For example, the `pacda` specifies signing data pointers using the `A` key.

Table 1: Summarized security goals and their corresponding security properties in full-path analysis.

| Security Goal | Description | Against Attacks | Corresponding Security Properties in Fig. 1 |
|---|---|---|---|
| **SG-1: Memory Safety** | ① Programs can only access BPF memory, and specific kernel objects such as context. | OOB Access | BPF object OOB I/II, kernel object OOB I/II, permission violation I/II, *type mismatch* |
| **SG-2: Information Leakage Prevention** | ① Programs cannot write pointers into maps, and calculation among pointers is not allowed. | Layout Leakage | pointer leakage I/II, offset leakage, *type mismatch* |
| | ② Programs cannot read uninitialized information. | Uninitialized Rd | uninitialized register read, uninitialized stack read I/II |
| | ③ Programs cannot speculatively access areas outside the BPF program's memory. | Spectre | Spectre V1 filter/masking, Spectre V4 barrier |
| **SG-3: DoS Prevention** | ① Programs cannot crash while executing. | Crash Kernel | kernel stack crash I, kernel stack crash II |
| | ② Programs cannot execute for too long. | Denial-of-Service | timeout, deadlock |

by-zero check checks if the `imm` operand in the instruction is 0. Filtering such checks is simple by inspecting whether the condition contains the `env` or not. This is due to the program state used in the state machine being recorded in the verifier's environment structure (`struct bpf_verifier_env *env`).

Among all the checks, some are complexity or functionality checks, such as the combined branch state cannot exceed `8,192` to prevent state explosion. We filter these checks by understanding their semantics, supplemented by the error message. For example, these checks usually contained phrases like "`too many`" or "`too complex`". Conversely, the security checks typically contained words like "`prohibited`" or "`invalid`". By understanding the filtered security checks and their semantics, we obtain all the security properties that the verifier wants to ensure in the full-path analysis.

### 3.2.2 The Security Properties in Full-path Analysis

Fig. 1 gives a panoramic view of how the verifier performs the security checks in the full-path analysis. `bpf_check()` is the entry point of the verifier, and the `do_check()` it calls contains most of the security checks in the full path analysis. Since they are performed during the path traversal along with the state tracking, we call them *inflight checks*. Whenever an instruction is encountered, it will perform security checks based on the semantics of the instruction and all operands' states (i.e., the type and the value range). After the path traversal, the `check_max_stack_depth()` and `convert_ctx_accesses()` are called in sequence to perform some global security checks, which we call the *landing checks*. For instance, it checks the combined BPF stack frames' size does not exceed the upper limit to prevent kernel stack crashes. Finally, `do_misc_fixups()` is called to instrument some sanitization code to prevent Spectre attacks.

The verifier contains hundreds of security checks, we have merged and simplified similar security checks in Fig. 1 for ease of introduction. The left and right sides give the summarized 20 security properties of these security checks. For example, underflow and overflow are both disallowed when accessing BPF objects, which corresponds to *BPF object OOB I*. As for kernel object access instructions, the immediate offset is checked based on a whitelist to ensure the BPF program only accesses legal fields, which ensures *kernel object OOB I*.

While some security checks have explicit security properties that can be summarized based on judgment conditions and error messages, some properties need to be inferred. Take *uninitialized stack read II* as an example, BPF programs are not allowed to write to the stack using variable offsets due to the state maintenance of stack slots. When a variable offset is involved and a stack slot is uninitialized, the verifier cannot determine if this write would initialize this slot, potentially leading to an uninitialized stack read. The detailed descriptions of all security properties are listed in Appendix A.

**The summarized security goals.** Security properties are the security that the verifier must ensure at the implementation level. Based on them, we also summarize the security goals at the design level, as shown in Table 1. All security goals were emailed to the eBPF developers and got confirmation. For **SG-1**, BPF programs can only access their own memory. The permission violation and type mismatch between pointers could lead to illegal memory access, thus they are also memory safety related; For **SG-2**, the information leakage should be prevented, including kernel layout, uninitialized information, and kernel memory accessed speculatively. Besides, the type mismatch between scalars and pointers could also leak kernel layout; For **SG-3**, DoS attacks should be prevented due to the preempt being disabled when the BPF program is running, and the exception handling is not supported in eBPF.

## 3.3 The Dilemma of Full-path Analysis

Although eBPF developers have been dedicated to improving the verifier, it still faces the following dilemmas.

**The capability dilemma.** To avoid the *state explosion* and ensure the analysis can finish in a constant time, several restrictions are imposed on the program, including limiting the number of loops and branches. Kernel developers have recognized this challenge and tried to address it over the years. For example, eBPF supports the intra-procedure analysis instead of the whole program in 2017. However, it can lead to state overestimation and even exacerbate complexity problems.

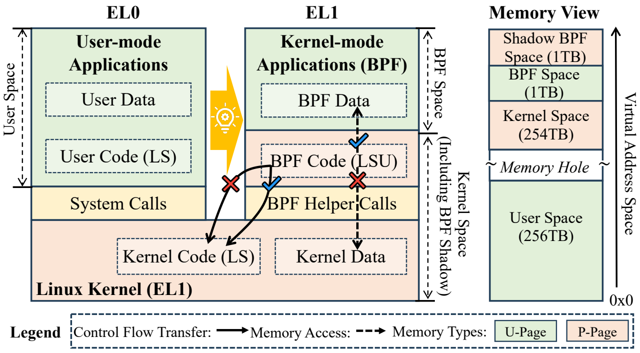


Fig. 2: The high-level design of Hive.

Constant loops were supported in 2019 after extensive design and attempts. Helper functions such as `bpf_loop()` were introduced in 2021 to handle a special kind of loops. However, while these functions seemingly address the loop problem, they impose limitations on passing state between loop iterations, along with the loop body and condition, deviating from commonly used loop patterns (detailed in Appendix B). Up to now, complex BPF programs still cannot pass the verification.

**The correctness dilemma.** The code size and complexity of the full-path analysis have significantly increased over the years, making formal verification challenging and leading to design and implementation bugs. The full-path analysis has been responsible for over 90% of CVEs in the eBPF verifier in the past decade. Despite its potential, unprivileged eBPF programs have been dismissed as unsafe by the Linux community. As the core of the verifier, the intricate implementation of the full-path analysis contributes to these security issues.

# 4 Overview

## 4.1 Threat Model

The goal of Hive is to provide an isolated execution environment for BPF programs to replace the full-path analysis in the verifier. We assume the pruned verifier and other components of the eBPF subsystem are secure and trustworthy. The goal of the adversary is to compromise the security goals listed in Table 1 through adversary-controlled BPF programs. BPF programs may execute arbitrary legal instructions, access arbitrary memory, and call arbitrary helpers. Note that the pruned verifier still guarantees the control flow safety.

## 4.2 High-level Design

Our key insight is that BPF programs can be executed as kernel-mode applications. An analogy can be drawn from the isolation of user applications: User applications running at EL0 cannot access kernel memory because the kernel memory pages are set up as privileged pages; each user application runs in its own independent and continuous address space; crash isolation can be achieved via monitoring exceptions. If an isolated execution environment can be created for BPF programs, some of the heavy-lifting tasks performed by static verification can be replaced by hardware-assisted dynamic isolation. This is the key idea behind Hive.

As shown in Fig. 2, Hive creates such an isolated execution environment as follows: *Firstly*, Hive creates an independent address space for each BPF program, dubbed *BPF space*. BPF space holds all BPF data and is set as U-Pages. Note that the BPF code is kept in the kernel space, which is set to be P-Pages, as is done in a vanilla OS. In this way, users cannot infer the kernel layout from the BPF space. *Secondly*, Hive emits all BPF memory access instructions as LSU instructions, which can access the BPF data on the U-Pages, but cannot access the kernel memory on the P-Pages. Because of Privileged Access Never (PAN) [6], helper functions are forbidden from accessing the BPF data. Therefore, Hive creates a separate space that maps the BPF data as P-Pages for them, hence creating a *shadow BPF space*. Both shadow BPF spaces and BPF spaces are 1TB in size and are placed at the highest address. *Thirdly*, Hive captures all exceptions and times the execution of BPF programs. Once an exception or a timeout is caught, Hive will roll back the state to the entry of the target BPF program and unload it.

## 4.3 Challenges

Since BPF programs are highly coupled to the kernel, isolating them from the kernel still faces the following challenges:

**C-1: BPF data requires object-granular isolation.** The full-path analysis tracks the type of each pointer and the range of the pointed object, ensuring that it can only access the *accessible region* inside the BPF data object. This is because BPF data objects are non-contiguous and interleaved with other inaccessible objects. For example, eBPF embeds object management structures (including function pointers, lock pointers, etc) into these objects, which also need protection. However, the EL-based memory isolation used in Hive cannot provide such finer-grained (or sub-page) protection.

**C-2: Kernel objects need to be accessed securely.** eBPF allows BPF programs to directly access specific fields of kernel objects. However, such fine-grained kernel object access control is difficult for two reasons: 1) The LSU instructions cannot access the kernel space; 2) Hive cannot distinguish which instruction accesses the kernel object. Simply setting the page where the kernel object is located to be U-Page will be vulnerable to sub-page attacks; Copying all accessible fields of kernel objects into the BPF space each time the BPF program is executed will bring high performance overhead.

**C-3: The pointers of kernel objects cannot be leaked.** Since maps can be accessed by users through system calls, the full-path analysis prevents all pointers from being written to maps,

Table 2: Differences between different eBPF pointer types.

| Types | Point to | *Modifiable* | *Instr. De-reference* | | *OOB Check Method* |
|---|---|---|---|---|---|
| | | | Access Form | Pinned | |
| Inclusive[1] | BPF obj | ✔ | arbitrary form | ✘ | bound-check |
| Exclusive[2] | Kernel obj | ✘ | constant offset[3] | ✔ | whitelist |

[1] Including 10 types: *ptr_to_stack*, *ptr_to_buf*, *ptr_to_mem*, *ptr_to_tp_buffer*, *ptr_to_map_key*, *ptr_to_map_value*, *ptr_to_packet*, *ptr_to_packet_meta*, *ptr_to_packet_end*, and *ptr_to_flow_keys*.

[2] Including 8 types: *ptr_to_tcp_sock*, *ptr_to_socket*, *ptr_to_sock_common*, *ptr_to_xdp_sock*, *ptr_to_ctx*, *ptr_to_btf_id*, *ptr_to_map*, and *ptr_to_func*.

[3] Except for *ptr_to_map* and *ptr_to_func*, other types of pointers are allowed to be de-referenced directly via memory access instructions.

preventing the leakage of pointers. HIVE places BPF's data into a fixed BPF space to eliminate security problems caused by the leakage of BPF object pointers. But, the pointers that point to allowed kernel objects must be prevented from being leaked. Without pointer/type tracking, HIVE cannot recognize these pointers and prevent them from being leaked.

# 5 Detailed Design

The above challenges are due to the lack of pointer/type tracking capabilities in HIVE. In this section, we analyzed the pointer type features of eBPF in §5.1, and introduce how HIVE integrates its design in §5.2, §5.3, and §5.4. Importantly, the design of HIVE is compatible with the eBPF standard, allowing seamless execution of any BPF program.

## 5.1 eBPF Pointer Types

There are 18 types of pointers in eBPF and we classified them into two types based on their usage and constraints: the *inclusive type* and the *exclusive type*. The *inclusive type* pointer points to the BPF data object, and the latter points to the kernel object. Pointers of the same type share the same functional constraints but differ between different types.

In eBPF, the *exclusive type* is performed with more strict constraints. This is due to the support of direct memory access to specific fields of kernel objects [1]. eBPF introduces a virtual data structure for each BPF-accessible kernel object during programming. Each virtual structure solely encompasses BPF-accessible fields of its corresponding kernel object. At load time, eBPF will relocate all accesses to the fields of virtual structures to the actual fields of kernel objects. However, the precise relocation requires three constraints (shown in Table 2): 1) *their value cannot be modified*, allowing for accurate pointers tracking; 2) *they can be only de-referenced through the memory access instruction in the form of pointer plus offset*, allowing for the relocation by rewriting the immediate operand of the instruction; 3) *memory access instructions that access a kernel object are not allowed to be used to access BPF objects or even other types of kernel objects*, allowing for enforcing the memory access control.

Except for the *OOB*-related security properties, all properties are the same for these two pointer types. The difference in OOB security properties is just in how they are ensured: *the OOB check for the inclusive type is a range check, but for the exclusive type, it is a whitelist based on the kernel object*.

## 5.2 Handling Inclusive Pointer Types

In eBPF, all the inclusive type pointers have the same security checks, thus they have a unified parent type `mem_type`. However, security checks are only performed by using the more exact inclusive types due to only raw data in BPF objects can be accessed and stored in fragments. This fragmented accessible memory poses challenges to the EL-based memory isolation. A feasible approach is to organize accessible segments into a contiguous BPF memory space.

### 5.2.1 BPF Memory Compartmentalization

In this section, we introduce how HIVE compartmentalizes all BPF objects into the BPF space.

**Stack separation.** BPF programs use a stack, which is on top of the kernel stack. The stack frames of BPF functions are only accessed through the BPF stack frame pointer register, which is mapped to a physical general-purpose register (the `%x25` on AArch64), and its derived pointers. The JIT compiler will also instrument code to store/restore (using the `%sp` register) return addresses and callee-saved registers during the BPF function call/return. In HIVE, the BPF stack frames are separated from the kernel stack by isolating them to a new stack. At load time, HIVE allocates a dedicated memory region as the BPF stack (for each CPU core) in the BPF space, and adjusts the `%x25` to point to it. As such, helper functions can be called freely without stack switching and still use the kernel stack. To prevent attackers from causing the kernel stack to overflow through continuous function calls, HIVE also places an unmapped guard page at the lowest position of each kernel stack. HIVE also instruments instructions to manage BPF stack frames (detailed in §C.1).

**Maps separation.** Maps are composed of metadata and values that are stored next to each other. The metadata usually contains the type, data pointers, and function pointers, which must be separated from values. To do this, HIVE slightly adjusts the structure of the map, separates all the values from the maps, maps them into the BPF space with the allowed permission (e.g., the read-only), and leaves a pointer pointing to them (detailed in §C.2). This change is transparent to the BPF program because it only uses the address of the value, and the parsing on the maps is done by helper functions.

**Packets separation.** Packets, which are buffers pointed to by the `data` field in the context, are allocated when network packets arrive. Simply mapping them into the BPF space could bring high-performance overhead due to the packet freeing will cause the mapping canceling, which incurs the high-cost TLB shootdown. To this end, HIVE allocates the packet from a dedicated memory pool, which is implemented as a `kmem_cache` in the slab. It will be double-mapped into the BPF space with allowed access permissions based on the BPF program's type, such as read-only and writable. This eliminates the need for high-cost page table manipulation.

Since a BPF program processes all packets sequentially (either individually or in groups), it is secure allowing access to all packets at the same time. Furthermore, for parallel packet processing, HIVE prevents potential concurrent attacks by isolating different queues. Specifically, BPF programs can utilize multi-queue network interfaces for parallel execution, these interfaces have multiple independent receive/send queues, and each queue will be handled in parallel on a different CPU core. HIVE associates each queue with specific `kmem_cache` and maps them into different BPF spaces of a BPF program respectively. When the BPF program processes the packets in a queue, HIVE will set the corresponding BPF space for the BPF program, so that the BPF program can only see all the packets in one queue at the same time.

### 5.2.2 BPF Memory Isolation (SG-1)

**Isolation of direct memory accesses.** HIVE separates all accessible BPF objects and maps them in the BPF space. Therefore, containing type pointers no longer requires object-level range checking, they only need to check whether the access target is in BPF space through the EL-based memory isolation mechanism. However, this method leads to a new problem: user programs and BPF programs can access each other's memory. To address this problem, HIVE leverages `E0PD` to enable/disable unprivileged access to user space and BPF space. Specifically, when the kernel calls a BPF program, HIVE will clear `E0PD1` to enable the access to the BPF space, and set `E0PD0` to disable the access to user space; when the kernel returns to the user program, HIVE checks if any BPF program has executed, if it has, HIVE reverse the `E0PD` settings. This design can avoid unnecessary `E0PD` settings.

**Sanitization of helper's parameters.** The inclusive type pointers can be passed to helpers through parameters. The full-path analysis performs the range/type checking for the parameters that HIVE should do the same. Besides, as mentioned in §4.2, the inclusive type pointers also need to be converted to point to the shadow BPF space when passed to helpers. Benefiting from the design of BPF space memory layout/alignment, the range/type checking, and the pointer conversion can be performed through one logical operation instruction. HIVE instruments an `orr xn, 0xffffff0000000000` instruction for each function parameter of the inclusive type pointer before calling helpers. In this way, the kernel can access the BPF space in a disguised way while the pointer is also checked. Note that an attacker may set a pointer point to the boundary of the BPF space to induce helpers to access memory outside the BPF shadow space. But it is safe that the helper will overflow access to BPF space and trigger permission fault.

### 5.2.3 Information Leakage Prevention (SG-2)

**Independent address space (SG-2.1).** HIVE creates an independent address space for each BPF program and ensures separated BPF objects are randomly mapped into it. All the inclusive type pointers are relocated to point to this space. Thus, attackers cannot infer any kernel layout information from the inclusive type pointers that can be safely leaked.

**Use after initialization (SG-2.2).** HIVE initializes all memory allocated in the BPF space at load time. Each time the BPF program is executed, the BPF stack of the current core will be reused without re-initialization. If an uninitialized read of the BPF stack occurs, it will still access the content of the last execution and will not cause memory leaks in the kernel.

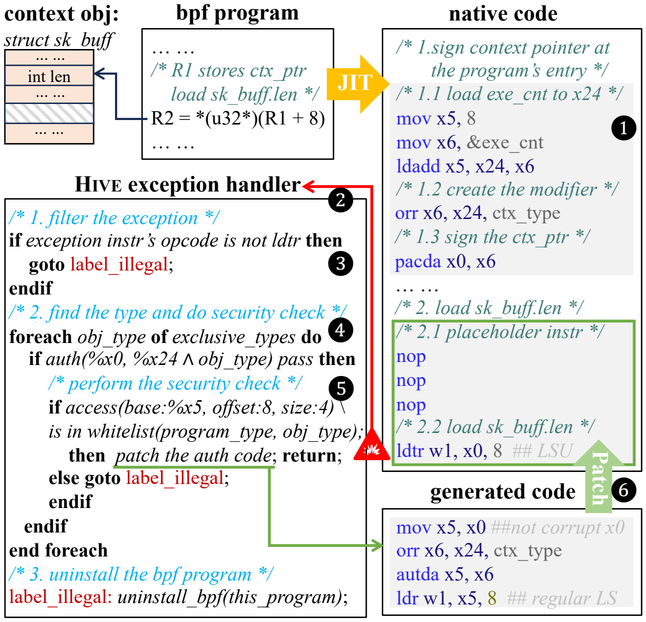
**Convert Spectre to Meltdown (SG-2.3).** The key to Spectre v1 and v4 attacks [21] is to transiently access the sensitive data and use its value to encode and access a legal memory address that could cause the change of the cache state. As mentioned in §4.2, BPF programs use LSU instructions in HIVE, thus accessing the kernel space (speculatively) will cause the permission fault. A hardware patch named CSV3 [5] is added to mitigate the Meltdown attack [27], which *forbids the data loaded under speculation with a permission fault to be used to form an address to be used by other instructions in the speculative sequence*. Therefore, thanks to the transformation of LSU instructions and the CSV3 patch, Spectre attacks can be blocked from the hardware level without any software effort.

## 5.3 Handling Exclusive Pointer Types

Since the exclusive type pointers cannot be modified and their de-referenced points are exclusive, we propose to *utilize the `PA` to ensure the pointers' integrity, and the regular `LS` (load/store) instruction to provide the ability to access kernel objects*. However, the `PA+LS` method still faces three challenges: 1) how to design a secure modifier to prevent pointer substitution attacks; 2) how to ensure the *authentication* operations are done at all points of use; 3) how to address the problem of information leakage. In the following subsections, we will introduce how HIVE addresses the above challenges.

### 5.3.1 The Design of Modifier and the Sign

PA suffers from pointer substitution attacks — pointers signed with the same modifier can be replaced with each other and pass the authentication [24, 25]. To avoid pointers being replaced by other type pointers, pointers from other BPFs, or pointers from historical executions, we must ensure that the modifier is *different for each exclusive type and historically*



```
… …
/* R1 stores ctx_ptr
   load sk_buff.len */
R2 = *(u32*)(R1 + 8)
… …
```

```
/* 1.sign context pointer at
     the program's entry */
/* 1.1 load exe_cnt to x24 */
mov x5, 8
mov x6, &exe_cnt
ldadd x5, x24, x6
/* 1.2 create the modifier */
orr x6, x24, ctx_type
/* 1.3 sign the ctx_ptr */
pacda x0, x6
… …
/* 2. load sk_buff.len */
/* 2.1 placeholder instr */
nop
nop
nop
/* 2.2 load sk_buff.len */
ldtr w1, x0, 8  ## LSU
```

```
/* 1. filter the exception */
if exception instr's opcode is not ldtr then
  goto label_illegal;
endif
/* 2. find the type and do security check */
foreach obj_type of exclusive_types do
  if auth(%x0, %x24 ∧ obj_type) pass then
    /* perform the security check */
    if access(base:%x5, offset:8, size:4) \
    is in whitelist(program_type, obj_type);
      then  patch the auth code; return;
    else goto label_illegal;
    endif
  endif
end foreach
/* 3. uninstall the bpf program */
label_illegal: uninstall_bpf(this_program);
```

```
mov x5, x0 ##not corrupt x0
orr x6, x24, ctx_type
autda x5, x6
ldr w1, x5, 8  ## regular LS
```

Fig. 3: An example of the exclusive pointer type solution.

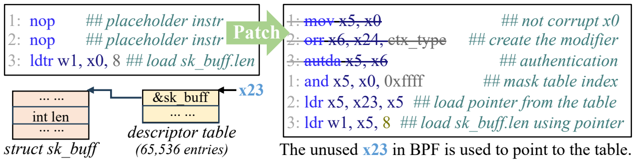
*unique to each BPF program*. Pointers of the same type generated during BPF program execution are safe to replace each other because the security checks are only type-related.

In HIVE, the modifier is designed to be the following form: `EXE_CNT` ∨ `PTR_TYPE`, where `EXE_CNT` is a 64-bit execution counter for all BPF programs. The counter only uses the upper 61 bits, which starts from 0 and increases by 8 each time, ensuring that the lowest 3 bits are 0. `PTR_TYPE` is a 3-bit constant value that represents the exact type (8 types in all). Since `EXE_CNT` is determined each time the BPF program is executed, HIVE uses the unused `%x24` register in eBPF to store its value at the program's entry. When signing or authenticating an exclusive type pointer, the OR result of the `%x24` and the type is calculated as the modifier (❶ in Fig. 3). Two unused registers `%x5` and `%x6` in eBPF are used as the temporal registers for HIVE to conduct such calculation.

For most cases, the `pacda` instruction is instrumented after the pointer generation point, such as relocation at load time, helper returns, and program's arguments. For the dynamically loaded pointers from kernel objects, HIVE instruments the `pacda` instruction right after the load operation at runtime because it can only be identified at runtime (detailed in §5.3.2).

### 5.3.2 The Authentication and Security Checks (SG-1)

The exclusive type pointers can be de-referenced directly and passed into helper functions. For each helper call, HIVE instruments an `autda` instruction to authenticate the parameter which is the exclusive type pointer. If authentication passes, it means the pointer has not been modified and matches the parameter type. However, it is difficult for direct memory accesses because HIVE does not know where to instrument.



```
1:  nop       ## placeholder instr
2:  nop       ## placeholder instr
3:  ldtr w1, x0, 8 ## load sk_buff.len
```

```
~~1: mov x5, x0~~                      ## not corrupt x0
~~2: orr x6, x24, ctx_type~~          ## create the modifier
~~3: autda x5, x6~~                    ## authentication
1: and x5, x0, 0xffff             ## mask table index
2: ldr x5, x23, x5  ## load pointer from the table
3: ldr w1, x5, 8  ## load sk_buff.len using pointer
```

The unused x23 in BPF is used to point to the table.

Fig. 4: The code de-referencing the exclusive type pointers.

**Exception-based point-of-use probing.** To address this, we observed that de-referencing a signed exclusive type pointer via the LSU instruction will trigger the hardware exceptions in HIVE (will be discussed later), so we can know the use points of the exclusive type pointers lazily by catching and filtering such exceptions (❷❸ in Fig. 3). It is feasible that once an instruction accesses one type of kernel object, it can no longer access other types of objects. Note that since the PAC is stored in the highest 16 bits of a pointer, and the highest 16 bits of the effective addresses in the kernel and BPF spaces are both 1, the sign operation will not change the pointer, that points to the kernel object originally, to point to the BPF space. The signed pointer may point to the user/kernel space or the memory holes, accessing them by using the LSU instructions will trigger the permission fault or the translation fault.

**The authentication code generation.** A BPF program usually has different exclusive types of pointers, and HIVE needs to know which type of de-referenced pointer triggers the exception. To this end, HIVE tries to use all types to authenticate the pointer when handling the exception (❹). Only when the types match can it pass the authentication. If no type is matched, it means that the pointer may be corrupted and HIVE will uninstall the BPF program. If a type is matched, HIVE will check the whitelist based on the range of access (from the instruction operand), the program type, and the exclusive type (❺). If the check passes, HIVE will generate a regular LS instruction along with the authentication code and patch the original code (❻). Note that the patched code cannot corrupt the signed pointer due to it may be used to probe subsequent use points. To achieve in-place patching, HIVE inserts `nop` instructions during the code jitting before memory access instructions with the constant offset that is in the whitelist.

### 5.3.3 The Design of Type Descriptor Table (SG-2)

All kernel objects accessed in BPF programs are initialized by the kernel (that ensures **SG-2.2**). But, the `PA+LS` method cannot ensure the confidentiality of pointers (**SG-2.1**), and the PA feature is vulnerable to Spectre attacks [33] (**SG-2.3**). Therefore, we need another method that is compatible with the `PA+LS` method to solve these problems. Inspired by the *file descriptor* mechanism, we propose a *type descriptor based* method to hide the real pointers and enforce their usage.

Specifically, HIVE transforms the pointer to a descriptor which indexes a descriptor table that stores the actual pointers in it. HIVE sets up a table with `65,536` ($2^{16}$) entries for each exclusive type. The base address of each table will be stored in an unused register in eBPF. Pointers to different objects of the same type are stored in different entries, and unused entries are filled with invalid addresses. Fig. 4 shows the patched code based on the descriptor table, which is used to replace the patched code based on authentication (❻ in Fig. 3). It performs a mask operation on the descriptor to ensure that the access to the descriptor table will not go out of bounds (line 1), then uses the descriptor to index the table of the corresponding type to obtain the actual pointer (line 2), and finally uses the regular `LS` instruction to access the kernel object (line 3). For helper calls, HIVE inserts the same instructions at lines 1-2 to recover the pointer for each exclusive type pointer parameter and store the original value in the kernel stack; when the helper returns, HIVE restores the original value of the parameter to prevent the pointer leakage. The design of the descriptor table ensures pointers' confidentiality (**SG-2.1**) and the mask operation prevents Spectre attacks (**SG-2.3**).

The authentication-based point-of-use probing method remains the same, we only replace the authentication-based patched code with the table look-up code. The newly patched code could also provide the same security guarantees on **SG-1** — *it forces the use of the table corresponding to the target type to ensure type matching and the addresses of target objects obtained from the table are either legal or invalid.*

### 5.4 Secure and Passive DoS Prevention (SG-3)

BPF program may bring two types of DoS: trigger an exception, and execute without terminating. For the former case, HIVE passively captures all triggered exceptions, rolls back the state to the entry point of the program, and unloads it; For the latter case, HIVE checks the BPF execution time each time there is an interrupt coming and rolls back the state when the executed instructions reach a certain threshold.

**Exceptions capturing.** The BPF program changes the memory state in three folds: BPF data in the BPF space, BPF management structures in the kernel, and some kernel objects. Changes to the first two types of data do not affect the state of the kernel. The fields modified by the third type of kernel object are just raw data for the kernel and do not affect the execution of the kernel. Therefore, they can be cleared by uninstalling the BPF program simply. HIVE only needs to store all BPF-accessible registers at the entry of the BPF program and restore them during the roll-back.

**Execution timing.** HIVE maintains a timetable for each executing BPF program to track their execution time. When an interrupt occurs, HIVE checks the duration of the currently executing BPF program and either uninstalls it if it exceeds the configured time limit or updates the timetable if it is within the limit. The advantage of this method is that it accurately accounts for the execution time of the BPF program, including the helper functions, which could avoid DoS problems caused by underestimating execution time in the verification [18].

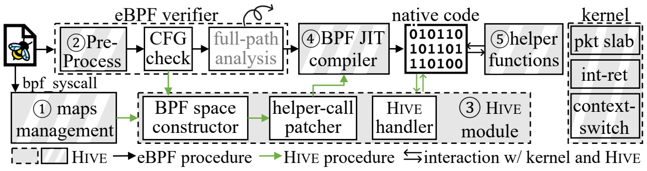


Fig. 5: The workflow of HIVE.

## 6 System Implementation

The prototype of HIVE (shown in Fig. 5) is implemented on Linux-v6.6.1/AArch64. The core of HIVE is a kernel module (③) to manage the BPF space, handle helper calls, and manage exceptions. HIVE modifies the eBPF subsystem and the core kernel. For the eBPF subsystem, HIVE modifies the map management (①) to achieve maps separation, the verifier (②) to skip the full-path analysis, the JIT compiler (④) to perform the code instrumentation and transformation, and rewrites 11 helper functions (⑤); for the core kernel, HIVE only modifies 43 LoC, including creating the packet slab, configuring the `E0PD`, and timing the execution of BPF programs.

**Patch detail.** During page fault handling, the patching handler is invoked with the fault type and address. To avoid concurrent execution, the handler's execution is protected by a *mutex* lock. Upon acquiring the lock, it verifies the LSU form of the patch point. If passed, an IPI is sent to halt other cores' execution, forcing them to execute a waiting function, which disables interrupts and repeatedly checks a patching flag. Subsequently, the handler performs the code patching by changing the code page to writable, carrying out the instruction rewriting, flushing the instruction cache, and changing the permission back. Finally, the handler sets the patching flag to resume execution on other cores and returns to re-execute at the fault location.

**Prevent reading of uninitialized registers.** For the 11 registers used for eBPF, the full-path analysis forbids the BPF program to read them before they are initialized. To this end, HIVE clears all possible uninitialized registers when the BPF program interacts with the kernel. Specifically, HIVE zeros all context-irrelevant registers at the program entry, and clears all caller-saved registers during the helper function returns.

**Register uses of HIVE.** On AArch64, 12 registers are unused in eBPF (`%x5-6`, `%x8`, `%x13-18`, `%x23-24`, `%x28`) and can be used in HIVE. HIVE uses 2 registers (`%x5-6`) as temporary registers, 6 registers (`%x13-18`) that store the base of descriptor tables (for a BPF program type, up to 6 exclusive types can be used simultaneously), 2 registers that store the frequently used pointers (`%x23` for the context, `%x24` for the `EXE_CNT`). When calling helper functions, HIVE does not need to store

Table 3: Security equivalence analysis between Hive and the verifier in the full-path analysis.

| Line | Security properties | How Hive ensures the corresponding security property | Equal |
|---|---|---|---|
| 1 | BPF object OOB I/II | Hive compartmentalizes all accessible areas of BPF objects into the BPF space, thus the object-granular OOB is relaxed to space-granular OOB securely. The use of EL-based memory isolation prevents BPF programs from the BPF space OOB. | ✔ |
| 2 | kernel object OOB I/II | Accesses to kernel objects are converted to use the descriptor table, which only contains accessible kernel objects. Besides, whether the accessed fields of accessible objects are legal is verified during the probe process. | ✔ |
| 3 | permission violation I | The compartmentalized maps' values and the packet slabs are mapped with the allowed permission into the BPF space. | ✔ |
| 4 | permission violation II | The first access to a kernel object field will be captured and checked for access permission in the probe process. | ✔ |
| 5 | pointer leakage I/II,<br>offset leakage | The independent BPF space eliminates the kernel layout information in inclusive type pointers that can be safely leaked. For exclusive type pointers, the type descriptors hide the real pointers and only indicate the use order of kernel objects. | ✔ |
| 6 | type mismatch | Hive matches the pointer type parameters at helpers call. For inclusive pointer types, Hive uses the `OR` operation to ensure the pointer points to the BPF space to achieve type matching. This is because Hive compartmentalizes all BPF objects into BPF space, thus these types can be safely cast to their unified parent type `mem_type`, which is equivalent to the BPF space; For exclusive pointer types, Hive enforces to use of the corresponding type descriptor table during the helper call to ensure type matching; For scalar type, Hive performs nothing because casting to a scalar from a pointer is safe since pointers have no kernel layout information anymore. Note the parameter type matching when calling between BPF functions is unneeded because it serves the state tracking in the intra-procedure analysis of the verifier and is not required by Hive. | ✔ |
| 7 | uninitialized reg read | The verifier checks the registers are initialized before use to prevent kernel registers leakage. Hive provides same guarantee by clearing all BPF-used registers at the program's entry and all non-caller-saved registers during the helper return. | ✔ |
| 8 | uninitialized stk rd I/II | BPF stack is decoupled with kernel stack and is initialized at program loading time. | ✔ |
| 9 | Spectre V1 filtering,<br>Spectre V1 masking,<br>Spectre V4 | The LSU instructions cannot be utilized to access the kernel speculatively due to the CSV3 patch. The LS instructions used in Hive are prevented from speculatively accessing kernel memory outside type descriptor tables and allowable kernel objects by using dedicated registers (to store the table base address) and masking the descriptor (to avoid the table OOB). | ✔ |
| 10 | kernel stack crash I | BPF program may crash the kernel stack by nesting function calls, Hive maps an unmapped guard page at the lowest position of kernel stack to avoid overflow via continuously pushing registers and return addresses on the kernel stack. | ✔ |
| 11 | kernel stack crash II | BPF program may crash the kernel stack by allocating a huge frame. Since Hive separates the BPF stack from the kernel stack, the frame size has no impact on the kernel stack—the size of the content Hive pushes on the kernel stack is fixed. | ✔ |
| 12 | timeout, deadlock | Hive times the execution of each BPF program, and it uninstalls the program if its execution exceeds the configured time. | ✔ |

`%x23` and `%x24` due to they are callee-saved registers, `%x13-18` will be protected by storing them in the kernel stack.

**Inter-BPF isolation.** Similar to the user space, the BPF space only holds one BPF program's data. Hive switches BPF space by switching the kernel page table (setting `TTBR1_EL1`) for different BPF programs. In detail, Hive allocates an ASID and creates a kernel page table for each BPF program which shares kernel space but has independent (shadow) BPF space. To avoid unnecessary switching, Hive merges BPF programs with the same type and capability from the same user into one BPF space and only switches page tables when different BPF programs switch executions.

**Additional helper functions handling.** The verifier maintains a list, which is reused by Hive, to ensure the BPF program can only call the allowable helper functions based on the program type. Among all 209 helper functions in Linux-v6.6.1, 198 helpers can be handled by the aforementioned methods to ensure argument safety. 11 helpers are discussed separately because the full-path analysis performs additional functional checks, and removing such analysis in Hive could affect the functionality. Take the `bpf_spin_lock()` as the example, the full-path analysis needs to ensure there is no spinlock acquired in the current program state. Hive rewrite such helpers by adding additional record-and-check logic in them (detailed in Appendix D).

**Security property customization.** BPF programs are specified with capabilities, and the verifier only ensures part of the security properties based on the specified capabilities. There are 4 types of capabilities in eBPF, i.e., `CAP_BPF`, `CAP_NET_ADMIN`, `CAP_PERFMON`, and `CAP_SYS_ADMIN`. For example, the verifier ensures all security properties for programs with only `CAP_BPF`, while do not consider the pointer leakage and Spectre for the BPF programs with `CAP_PERFMON` or `CAP_SYS_ADMIN`. Similarly, the security properties guaranteed by Hive can also be customized. For example, Hive can use the *authentication-based* method instead of the *descriptor table* method (§5.3.3) on the patched code for that capability combination. Now, Hive supports all security properties by default, we leave the customization as future work.

## 7 Security Evaluation

In this section, we not only analyze the security of Hive but also use real attacks to evaluate its security effectiveness.

### 7.1 Security Analysis of Hive

Hive brings two types of attacks: attacks on the security goals of the full-path analysis, and attacks on Hive framework.

**Security equivalence analysis.** In §5, we introduced how Hive achieves the security goals. Here we analyze how Hive ensures the security properties of the full-path analysis at the implementation level. Table 3 shows the point-to-point analysis, and the results illustrate that Hive can achieve equivalent security. In sum, Hive compartmentalizes and maps BPF objects with allowed permission into the isolated and

Table 4: Real attacks against the security properties.

| CVE ID | Root Cause | Target Property | Status[1] |
|---|---|---|---|
| 2020-27194 | Incorrect bound of `OR` insn. | dead loop | ● |
| 2021-3490 | Incorrect 32-bit bound of bitwise. | BPF obj OOB | ● |
| 2021-31440 | Incorrect bounds of 32-64 convert. | pointer leakage | ● |
| 2022-23222 | Mischeck of `*_OR_NULL` Pointer. | kernel obj OOB | ● |
| 2020-8835 | Incorrect 32-bit Bound. | kernel stack crash | ● |
| 2021-4204 | Improper input validation. | offset leakage | ● |
| 2023-2163 | Incorrect branch pruning. | type mismatch | ● |
| 2021-34866 | Lack map pointer validation. | permission violation | ● |
| 2021-33624 | Mispredicted branch speculation. | Spectre V1 | ❍ |

[1] ●: the attack is mitigated by HIVE, ❍: CVE is confirmed but lacks exploit.

address-independent BPF space to ensure BPF object (pointers) related properties (lines 1/3/5/6/9/10); HIVE proposes the exception-based probing and the type-based descriptor table mechanisms to ensure kernel object (pointers) related properties (lines 2/4/5/6/9); HIVE initializes the BPF memory and clears the BPF-used registers during the helper function calls to ensure uninitialized read related properties (lines 7/8); HIVE captures all exceptions and timing the execution time to ensure the DoS-related properties (lines 11/12).

**Particular attacks against HIVE.** We then discuss the possible attacks on HIVE architecture as follows:

- *Probe abusing attacks*: The probing mechanism allows kernel object access by transforming the `LSU` to the `LS` dynamically. Attackers may abuse it to access arbitrary kernel memory or trigger DoS. However, such an attack cannot be conducted: 1) HIVE verifies the legality of kernel objects at the first access time. The patched code forces access to legal objects and fields; 2) Although the probe triggers an exception, consuming more CPU time, the kernel will not be DoSed because HIVE also times the execution of the program, which will be uninstalled once timeout. Based on our experiments on a large number of BPF programs (§8), the kernel object access is concentrated (the probe count is in single digits), and legitimate BPF programs will not be unloaded due to the probe.
- *Descriptor replacement attack*: In HIVE, the authentication is only performed on the first access of the descriptor, which may allow attackers to corrupt the descriptor at subsequent accesses. However, HIVE ensures that each descriptor access point will only use a fixed and same type of descriptor table and the descriptor will be masked to ensure that it does not overflow the table. Therefore, even if the descriptor is modified, the attacker cannot access the kernel object (field) that is not allowed.
- *Attacks on PA*: Attackers may substitute exclusive type pointers that are signed with the same modifier [24, 25]. Due to the historical unique modifier design in HIVE, such an attack can only be conducted by massively executing BPF programs to overflow the 61-bit `EXE_CNT`. However, even the simplest BPF program with only prolog/epilog costs 129 cycles at least, $2^{61}$ executions would cost almost 3,000 years ($129 * 2^{61}/3.2Ghz/3600/24/365 = 2,947.56$), making this attack infeasible.
- *Hardware configuration tampering*: Attackers may directly disable LSU or `E0PD` through hardware configurations. For example, one may disable LSU by enabling the `UAO` bit in the `PSTATE` register or disable `E0PD` by setting the `TCR_EL1` register. However, both registers can only be modified via privileged instructions, which cannot be generated by the eBPF JIT compiler. Since the jitted code is protected by the DEP mechanism, the BPF program cannot inject such instructions via code self-modification.

```
1: /* precondition, R: R1 = 1, V: R1 lower
2:     32-bit = 1,  upper 32-bit unknown */
3: R2 &= R1 // trigger bug, R: R1=1, V: R1=0
4: R2 = <OOB off> // R: R2 = off, V: R2 = off
5: R2 *= R1 // R: R2 = off, V: R2 = 0
6: ptr += R2
7: *ptr = 0xbad // OOB access
```

(a) Code snippet of CVE-2021-3490

```
1: /* precondition, R1 stores ctx_ptr */
2: // R0: ctx_ptr, ctx_id in Hive
3: R0 = R1
4: if (R1->len > 4)
5:     // R0: sock_ptr, sock_id in Hive
6:     R0 = bpf_sk_fullsock(…)
7: R2 = *(u32*)(R0 + 0x4) // id replace
```

(b) Descriptor id replacement attack

Fig. 6: Code snippet of the different attacks. For (a), **R** denotes runtime value. **V** denotes verifier-deduced values.

## 7.2 Real Attacks against HIVE

To evaluate the security of HIVE, we investigated 9 CVEs (shown in Table 4) in the verifier and conducted attacks against specific security properties. To better explain how HIVE mitigates these CVEs, we elaborate on the exploit paths for CVE-2021-3490, which is a vulnerability due to the incorrect 32-bit boundary deduction for bitwise operations. As shown in Fig. 6 (a), it first constructs a register with the lower 32-bit value is 1 and the upper bit 32-bit value unknown, while the runtime value is 1. Then it uses an `AND` operation to trigger the bug, making the verifier mistakenly think `R1` is 0 while the runtime value is 1 (line 3). After the multiple and plus operation, the `ptr` is OOB, while the verifier still thinks it points to the beginning. The confused value allows it to bypass the verification for an illegal store (line 7). After HIVE transformed the memory access instruction into LSU, the store triggered a permission fault. Then, HIVE caught the fault and tried to authenticate the pointer. Since the pointer was forged, it failed authentication, leading to program uninstallation.

As for the Spectre attack of CVE-2021-33624, the verifier's Spectre mitigation can be bypassed because the verifier mispredicts the impossible branch. While its PoC was implemented on an x86_64 machine, we were unable to perform it on the AArch64 architecture. For other attacks, experiments showed that HIVE can successfully mitigate them.

To evaluate attacks against the HIVE framework, we wrote a BPF program to conduct a *descriptor replacement attack* (as mentioned before). Fig. 6 (b) gives the PoC that sets `R0` as different exclusive type pointers on different paths, which is illegal. Suppose on the first execution the `R0` is *ctx_ptr*,

Table 5: Latency (in cycles) of basic operations in HIVE.

| | **1st Set (*Absolute Value*)** | | | | **2nd Set (*Absolute Value*)** | | |
|---|---|---|---|---|---|---|---|
| **Ops** | *ldr* | *ldtr* | *str* | *sttr* | *pacda* | *autda* | *probing* |
| **Cycles** | 0.34 | 0.34 | 0.50 | 0.50 | 7.01 | 7.01 | 59.9K |
| | **3rd Set (*Difference Value*)** | | | | **4th Set (*Absolute Value*)** | | |
| **Ops** | *1-in* | *1-ex* | *1-in & 1-ex* | *2-in & 2-ex* | *E0PD* | *PT* | *REG_INIT* |
| **Cycles** | 8.60 | 8.62 | 8.72 | 8.82 | 48.12 | 30.06 | 11.03 |

Table 6: lmbench latency (in $\mu$s) and HIVE slowdown.

| **Config** | **null call** | **null I/O** | **stat** | **slct TCP** | **sig hndl** | **fork proc** | **Mmap Latency** | **Page Fault** |
|---|---|---|---|---|---|---|---|---|
| **Native** | 0.13 | 0.15 | 0.27 | 2.21 | 0.72 | 112.49 | 24.0K | 0.22 |
| **HIVE** | 0.15 | 0.18 | 0.31 | 2.34 | 0.81 | 125.12 | 24.8K | 0.23 |
| **Slowdown(%)** | 23.2 | 20.9 | 14.5 | 5.9 | 12.4 | 11.2 | 3.7 | 3.8 |

after the probe, the jitted code for line 7 is authenticated and rewritten into context-type-related descriptor code, which loads the context pointer from its corresponding table and performs the load operation. For the subsequent execution, no matter which path it takes, it always performed the context load operation, making the replacement attack ineffective.

## 8 Evaluation

**Experiments setup.** This section focuses on performance and complexity evaluation. The experiments are conducted on Linux v6.6.1, which runs on a Mac mini with an 8-core M1 CPU with PA support and 16 GB RAM. The processor is equipped with 4 efficiency cores and 4 performance cores, and all experiments were conducted on performance cores.

## 8.1 Performance Evaluation

### 8.1.1 Micro-benchmark

HIVE makes four sets of transformations on a program: **S1**: handling of inclusive pointer types; **S2**: handling of exclusive pointer types; **S3**: handling of helper function parameters; **S4**: preparing the environment at the program's entry. Therefore, we evaluated their latency separately (shown in Table 5).

For **S1**, we measured the latency of LSU instructions and found they are as fast as regular instructions (shown in column *1st Set*), which implies a high isolation efficiency.

For **S2**, we measured the instructions that related to the sign and the authentication operations in PA and found they are relatively efficient (shown in column *2nd Set*). We also evaluated the latency of one point-of-use probing and found that the overhead of probing a single kernel object access point is extremely high. But this is not a problem, because the subsequent experiments show that all probings can be completed in the first few executions of the BPF program.

For **S3**, we measured the instructions that handle helper parameters. We use an empty helper with different parameters (column *3rd Set*) to calculate the latency difference of a call with and w/o HIVE. The `1-in/ex` represents one parameter of inclusive/exclusive pointer type. The latency is relatively small and does not depend much on the number of parameters. The reason is that the handling between parameters has no data dependence, thus processors can process it in parallel.

For **S4**, we measured the latency of switching `E0PD` (setting `TCR_EL1`), switching page table (setting `TTBR1_EL1`), and dedicated registers initialization. As a comparison, we also evaluated the `SVC` instruction for syscall, whose latency is 145.37 cycles. We can see that these three operations are relatively efficient. Since the domain-switching affects the whole system state, we also evaluated its impact on the kernel operations. We loaded an empty BPF program to hook all syscalls and ran the lmbench [30]. The results (in Table 6) show HIVE incurs significant overhead on *null call* and *null I/O*, which only involve very little kernel operation. As a result, syscalls with simpler operations tend to have higher overheads.

### 8.1.2 Real-world Applications

To evaluate the overhead on real-world applications, we chose to load the BPF program in advance and then ran the applications to trigger the execution of the BPF program.

**Workload.** We chose two databases (Redis-7.2.4 and Memcached-1.6.21) and two web servers (Nginx-1.24.0 and Apache-2.4.59) as user applications. The experiments were conducted in the *local environment* to eliminate the network latency and evaluated with three configurations: without running BPF programs, running BPF programs with vanilla eBPF, and running BPF programs with HIVE-equipped eBPF. Web servers are evaluated with *ApacheBench* which sent a total of 100K requests with different request file sizes varied from 32KB to 256KB. Databases are evaluated with the *memtier* which sent 100K requests with different key sizes varied from 32B to 256B.

**BPF program selection.** The BCC project [35] provides a collection of BPF programs for profiling, performance analysis, network analysis, and security. We chose 32 programs and loaded them into the kernel by running their corresponding scripts at the same time. The Tracee [4] application contains 129 BPF programs for syscall tracing, which are loaded together by running the Tracee executable file. These BPF programs cover the majority of contemporary BPF ecosystem usage scenarios and are also used by other related works [28,29].

**Performance results.** We measured the throughput of the applications to evaluate the performance overhead brought by BPF programs. Table 7 gives the experimental results. The *baseline* column shows the throughput without BPF programs. The *eBPF-Tracee/BCC* and HIVE-*Tracee/BCC* columns show the throughput under vanilla eBPF and HIVE, respectively. The high overheads indicate that BPF programs are triggered

Table 7: The experimental results of real-world applications when running BPF programs with and w/o HIVE.

| App. | config | baseline | | eBPF-Tracee | | | eBPF-BCC | | | HIVE-Tracee | | | HIVE-BCC | | | HIVE/eBPF-O/H[4] | | exe_cnt/req[5] | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | THRU[1] | %CPU[2] | THRU[1] | O/H[3] | %CPU[2] | THRU[1] | O/H[3] | %CPU[2] | THRU[1] | O/H[3] | %CPU[2] | THRU[1] | O/H[3] | %CPU[2] | Tracee | BCC | Tracee | BCC |
| **Apache** | **32KB** | 18.50 | 98.6 | 10.48 | 76.6 | 98.4 | 6.17 | 199.9 | 99.1 | 10.11 | 82.9 | 98.6 | 6.03 | 206.9 | 99.1 | 3.48 | 2.28 | 555.1 | 568.8 |
| | **64KB** | 16.17 | 98.9 | 8.80 | 83.8 | 99.0 | 5.32 | 203.9 | 98.9 | 8.54 | 89.5 | 98.9 | 5.27 | 206.9 | 98.6 | 3.02 | 0.99 | 654.1 | 693.3 |
| | **128KB** | 12.52 | 99.0 | 6.65 | 88.3 | 99.0 | 3.60 | 248.1 | 99.1 | 6.42 | 95.0 | 99.4 | 3.46 | 262.2 | 98.4 | 3.46 | 3.90 | 809.6 | 1028.6 |
| | **256KB** | 7.70 | 99.6 | 4.41 | 74.6 | 98.5 | 2.01 | 282.2 | 98.1 | 4.26 | 80.8 | 98.5 | 2.01 | 282.8 | 98.1 | 3.44 | 0.16 | 1171.5 | 1749.5 |
| | **Geomean** | - | - | - | 80.6 | - | - | 231.1 | - | - | 86.9 | - | - | 237.4 | - | **3.34** | **1.08** | **766.1** | **917.9** |
| **Nginx** | **32KB** | 27.25 | 99.0 | 13.94 | 95.5 | 99.3 | 5.52 | 393.8 | 100.0 | 13.41 | 103.3 | 99.3 | 5.42 | 402.7 | 99.9 | 3.82 | 1.77 | 481.3 | 701.7 |
| | **64KB** | 23.96 | 99.0 | 12.34 | 94.1 | 99.5 | 4.48 | 434.8 | 99.9 | 11.86 | 102.1 | 99.8 | 4.40 | 444.8 | 99.8 | 3.95 | 1.83 | 584.6 | 823.9 |
| | **128KB** | 19.95 | 99.4 | 9.07 | 119.9 | 99.5 | 3.30 | 505.3 | 99.6 | 8.67 | 130.0 | 99.5 | 3.25 | 513.1 | 99.8 | 4.37 | 1.28 | 761.9 | 704.6 |
| | **256KB** | 12.98 | 93.4 | 5.85 | 121.8 | 99.5 | 2.26 | 474.9 | 98.0 | 5.58 | 132.5 | 99.0 | 2.19 | 492.5 | 99.5 | 4.60 | 2.97 | 1089.0 | 1912.4 |
| | **Geomean** | - | - | - | 107.1 | - | - | 450.2 | - | - | 116.1 | - | - | 461.2 | - | **4.18** | **1.87** | **695.1** | **939.5** |
| **Memc-ached** | **32B** | 1584.39 | 98.5 | 941.77 | 68.2 | 99.3 | 471.06 | 236.3 | 99.9 | 907.77 | 74.5 | 99.4 | 459.56 | 244.8 | 99.9 | 3.61 | 2.44 | 8595.7 | 13117.5 |
| | **64B** | 1583.11 | 98.6 | 939.88 | 68.4 | 99.3 | 467.08 | 238.9 | 99.9 | 906.88 | 74.6 | 99.4 | 458.95 | 244.9 | 99.8 | 3.51 | 1.74 | 8602.8 | 13110.0 |
| | **128B** | 1577.85 | 98.4 | 938.74 | 68.1 | 99.8 | 464.41 | 239.8 | 99.8 | 906.19 | 74.1 | 99.5 | 452.39 | 248.8 | 99.5 | 3.47 | 2.59 | 8647.7 | 13119.9 |
| | **256B** | 1551.61 | 98.6 | 923.09 | 68.1 | 99.5 | 461.82 | 236.0 | 99.6 | 883.12 | 75.7 | 99.3 | 455.12 | 240.9 | 99.6 | 4.33 | 1.45 | 8685.5 | 13115.6 |
| | **Geomean** | - | - | - | 68.2 | - | - | 237.7 | - | - | 74.7 | - | - | 244.8 | - | **3.71** | **2.00** | **8632.9** | **13115.8** |
| **Redis** | **32B** | 1342.35 | 88.7 | 861.30 | 55.9 | 90.0 | 698.98 | 92.0 | 66.7 | 836.33 | 60.5 | 81.0 | 689.23 | 94.8 | 67.9 | 2.90 | 1.39 | 975.9 | 1088.0 |
| | **64B** | 1304.76 | 100.0 | 861.96 | 51.4 | 81.7 | 663.63 | 96.6 | 65.7 | 836.59 | 56.0 | 82.0 | 659.54 | 97.8 | 64.6 | 2.94 | 0.62 | 1028.6 | 1399.3 |
| | **128B** | 1300.93 | 90.0 | 858.71 | 51.5 | 82.0 | 664.15 | 95.9 | 66.1 | 827.77 | 57.2 | 79.3 | 657.55 | 97.8 | 69.8 | 3.60 | 0.99 | 1020.9 | 1398.1 |
| | **256B** | 1292.59 | 90.0 | 855.05 | 51.2 | 90.0 | 656.88 | 96.8 | 70.0 | 821.67 | 57.3 | 80.0 | 652.03 | 98.2 | 68.0 | 3.90 | 0.74 | 1015.0 | 1408.2 |
| | **Geomean** | - | - | - | 52.4 | - | - | 95.3 | - | - | 57.7 | - | - | 97.2 | - | **3.31** | **0.89** | **1009.9** | **1315.8** |

[1] The application’s throughput (thousands of requests per second). [2] The CPU utilization (%). [3] The overhead (%) of vanilla eBPF and HIVE compared to baseline which does not load BPF programs.
[4] The overhead (%) of HIVE compared to the vanilla eBPF, which is calculated using the throughput directly. [5] The average number of times BPF programs are executed per request.

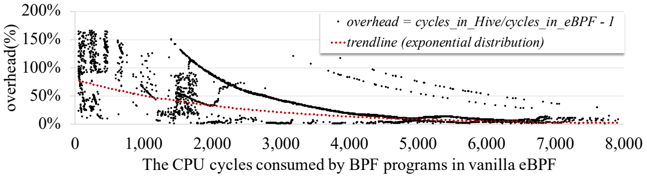


Fig. 7: HIVE’s overhead on the CPU time for BPF programs.

extensively. The HIVE/*eBPF-OH* column shows the overhead of HIVE compared with vanilla eBPF. On average, HIVE decreases the throughput by up to 4.18% and 2.00% when running Tracee and BCC compared to vanilla eBPF, respectively. It shows that HIVE is very efficient in practical applications even for such an extreme scenario of eBPF (i.e., running a huge number of BPF programs simultaneously).

To demonstrate that HIVE is fully evaluated, we also count CPU utilization and BPF program execution frequency. The results show that the CPU is saturated and the execution of the BPF program has little impact on the CPU utilization, except for Redis when running BCC. This is because Redis is a single-thread server and each BCC program has its independent process, causing process switching frequently. On average, each request triggered 695.1 to 13115.8 executions of BPF programs, showing that BPF programs were executed on the critical kernel path during the experiment.

**Statistic of probing times.** Since each probing will trigger an exception that consumes more CPU time, we performed statistics on point-of-use probes. The results show that a single probe consumes about 60k CPU time, and the number of probes of each BPF program is **less than 3 times**, which shows that the access to kernel objects is very concentrated.

**Statistic the CPU time consumed by BPF programs.** We counted the execution time of the same execution of BPF programs with or w/o HIVE. The determination method for the same execution under different configurations is the same BPF program, program parameters, and helper calling sequence at the same time. As Fig. 7 shows, when the BPF program’s execution time is relatively short (e.g., consuming less than 2,000 CPU cycles in vanilla eBPF), HIVE can bring up to 165% overhead. With the BPF program executing longer, the overhead decreased (e.g., less than 5% when vanilla eBPF consumes over 6,000 CPU cycles). This is because most of the overhead comes from the entry and exit code (e.g., domain-switching), while the overhead from instrumentation in the program’s body (e.g., register masking) is negligible.

## 8.2 Complexity Evaluation

As kernel developers claimed, the ultimate goal of eBPF is to “replace *kernel modules* as the de-facto means of extending the kernel”. Therefore, we used real-world kernel modules to evaluate the complexity promotion of HIVE. We also used a system call speed-up scenario to show the potential of HIVE.

### 8.2.1 Supporting Real-world Kernel Modules

We compiled 10 in-tree kernel modules, i.e., `polynomial`, `crc-ccitt`, `libarc4`, `prime_numbers`, `ghash`, `sha3`, `xxhash`, `libchacha`, `libsha256`, and `des`, from crypto utilities and general libraries into BPF programs. And we loaded them with the vanilla or HIVE-equipped eBPF. The vanilla eBPF rejected all of them due to non-constant loops (7), too many branches (1), or the computation of inclusive type pointers (2). Since loops and branches are no longer limited, and the calculation of inclusive type pointers is allowed w/o leaking the kernel layout, HIVE can successfully load and run these BPF programs. We obtained their output by using a user program to read the maps, compared them with that of the kernel modules, and confirmed the correctness.

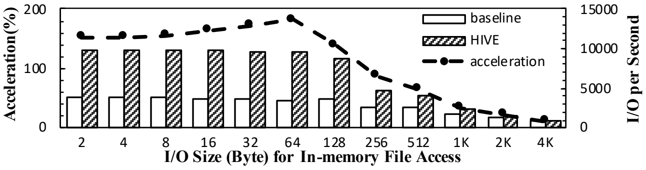


Fig. 8: IOPS (I/O per second) of `pread()` (baseline) and BPF program run in HIVE against different buffer sizes. The percentage number is the acceleration ratio of HIVE.

#### 8.2.2 Eliminating Context Switches in System Calls

User-kernel context switching often causes prominent overhead. Existing literature eliminated such overhead by running the syscall-intensive component in the kernel [22, 39]. The eBPF was not adopted because complex code could not pass the verification. However, this is no longer a problem in HIVE.

We followed the experiment used in UB [39] that accelerates the file I/O requests. The original program iteratively reads a file to a userspace buffer via `pread()` syscall. We compiled the syscall-intensive code into a BPF program and loaded it into the kernel. The syscall invoking was transformed into helper calling. The BPF program and the user program communicated through maps and a dedicated syscall. The vanilla eBPF rejected the BPF program due to the non-constant loop whose iterations number is passed from the user program, while HIVE can run it successfully. We gradually increased the buffer size for each read and evaluated the acceleration ratios. As Fig. 8 shows, HIVE accelerated syscall-based I/O up to 181% when the I/O size is small. For larger I/O sizes, IOPS acceleration drops due to the file accessing time becoming the dominant factor rather than context-switching.

## 9 Discussion

**BPF-infrastructure abusing attacks on HIVE.** While our threat model assumes a secure kernel, it is important to consider the implications if the kernel has vulnerabilities. As a representative work that abuses BPF infrastructure, EPF [19] uses the BPF interpreter and JIT compiler for privilege escalation attacks. In eBPF which is equipped with HIVE, EPF can still conduct such attacks by modifying the interpreter and the JIT compiler to inject gadgets. The defenses proposed by EPF and HIVE provide complementary approaches to isolate the kernel and BPF bidirectionally.

The Spectre V2 attack can exploit unprivileged eBPF to inject gadgets into the kernel to access the kernel memory speculatively [8]. HIVE transforms most of the memory access instructions into LSU form which cannot be used to leak the kernel data speculatively, but a very few regular memory access instructions are still used to access the kernel objects which could be abused by attackers: *they could control the table base register (%x23 in Fig. 4) to point to an arbitrary location speculatively*. However, it could be mitigated by transforming the code to use the PC-relative addressing form to enforce accessing the descriptor table.

Since the kernel has arbitrary access to the shadow BPF space, this makes "return-to-bpf" attacks possible: *the kernel can de-reference a corrupted data pointer to access the payload stored in the shadow BPF space*. This attack is similar to the classic return-to-user attack [20]. The difference is that the payload of the latter is stored in user space, so we could reuse the existing PAN-based protection mechanism against return-to-user attacks in Linux. Specifically, HIVE disables the shadow BPF space. Since the BPF space is set to be the U-pages, the kernel code cannot access it due to the PAN mechanism. Legal access of helpers can be through using the LSU instructions or disabling PAN temporarily.

**Applied to the X86 platform.** Since X86 does not have LSU-like instructions, we cannot de-privilege BPF memory accesses. The domain-based isolation hardware PKS [16] can be used to isolate the BPF program and the kernel in different domains and switch domains when accessing kernel objects or calling helper functions. PA technology can be implemented in software at low cost because HIVE only performs the *sign* and *auth* operations when the exclusive type pointers are generated and used for the first time. Since X86 has a smaller number of general-purpose registers (only 2 unused), while HIVE needs to use up to 10 registers, we can use floating point registers as "swap" registers.

## 10 Related Work

**Enhancing eBPF security.** Many works noticed the security issue and tried to harden the execution environment of BPF programs by using the Intel PKS [28] or SFI methods [26]. These works only considered memory safety and failed to harden all security properties which also exposed vulnerabilities. Even for memory safety, their security is weak due to the lack of complete BPF memory compartmentalization and object-granular protection of kernel objects. For example, they don't separate the metadata in maps, and the memory pages where the kernel objects are located are accessible.

**Use of load/store unprivileged instructions.** The LSU instructions have already been used for security [7, 10, 23, 38]. However, these works differ from HIVE in both scenario and design. To protect the system on ARM Cortex-M processors, Silhouette [38] and uXOM [23] transform regular load/store instructions to LSU instructions to protect the shadow stack and the code, respectively. uSFI [7] enforces the trusted code to use LSU instructions to access all memory. To protect the system on ARM Cortex-A processors (especially on AArch64), ILDI [10] and PANIC [36] protect the sensitive data in the kernel or user process by using PAN+LSU — the data are placed into U-Pages, and regular load/store instructions at EL1 cannot access them due to the PAN feature. They can only be accessed legally by using LSU instructions. Different from ILDI and PANIC, HIVE proposes to use LSU instructions to de-privilege the memory accesses of malicious BPF programs, thus protecting the trusted kernel. Especially, HIVE uses the EL-based memory isolation not the PAN.

**Use of pointer authentication.** PA has been used to protect the pointers in vulnerable software. PACStack [24] and PACTight [17] ensured the integrity of code pointers against the control flow hijacking attacks. PARTS [25] protected all code and data pointers. PTAuth [12] used PA to attach some information to data pointers to ensure the temporal memory safety of the heap. Differing from these works, HIVE uses PA to protect the pointers against malicious BPF programs. The biggest difference is that during the compilation phase, HIVE does not know at which points to authenticate.

## 11 Conclusion

The use of a static verification method in eBPF limits the complexity of BPF programs and brings security vulnerabilities. This paper presents a security-equivalent isolated execution environment on AArch64 for running BPF programs, which ensures kernel security at runtime and allows complex BPF programs to be loaded with low runtime overhead.

## Acknowledgments

We thank the anonymous reviewers for their insightful suggestions and comments. This research was supported by the National Natural Science Foundation of China (NSFC) under Grants 62272442, 61902374, U1736208, and the Innovation Funding of ICT, CAS under Grant No.E161040.

## References

[1] Alexei Starovoito. allow extended BPF programs access skb fields. https://lwn.net/Articles/636647/, 2015.

[2] Nadav Amit and Michael Wei. The design and implementation of hyperupcalls. In *USENIX ATC*, 2018.

[3] Andrew Werner. verifier escape with iteration helpers (bpf_loop, ...). https://lore.kernel.org/bpf/CAEf4BzZ-NGiUVw+yCRCkrPQbJAS4wMBsT3e=eYVMuintqKDKqg@mail.gmail.com/T/, 2023.

[4] Aqua. Aqua Tracee: Runtime eBPF threat detection engine. https://www.aquasec.com/products/tracee/, 2024.

[5] Arm Limited. Whitepaper Cache Speculation Side-channels, 2020.

[6] Arm Limited. Arm A-profile A64 Instruction Set Architecture, 2023.

[7] Zelalem Birhanu Aweke and Todd Austin. usfi: Ultra-lightweight software fault isolation for iot-class devices. In *DATE*, 2018.

[8] Enrico Barberis, Pietro Frigo, Marius Muench, Herbert Bos, and Cristiano Giuffrida. Branch history injection: On the effectiveness of hardware mitigations against Cross-Privilege spectre-v2 attacks. In *USENIX Security*, 2022.

[9] Ashish Bijlani and Umakishore Ramachandran. Extension framework for file systems in user space. In *USENIX ATC*, 2019.

[10] Yeongpil Cho, Donghyun Kwon, and Yunheung Paek. Instruction-level data isolation for the kernel on arm. In *DAC*, 2017.

[11] eBPF.io authors. Dynamically program the kernel for efficient networking, observability, tracing, and security. https://ebpf.io/, 2023.

[12] Reza Mirzazade farkhani, Mansour Ahmadi, and Long Lu. PTAuth: Temporal memory safety via robust points-to authentication. In *USENIX Security*, 2021.

[13] Luis Gerhorst, Benedict Herzog, Stefan Reif, Wolfgang Schröder-Preikschat, and Timo Hönig. Anycall: Fast and flexible system-call aggregation. In *Proceedings of the 11th Workshop on PLOS*, 2021.

[14] 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 *PLDI*, 2019.

[15] Yi He, Zhenhua Zou, Kun Sun, Zhuotao Liu, Ke Xu, Qian Wang, Chao Shen, Zhi Wang, and Qi Li. Rapidpatch: Firmware hotpatching for real-time embedded devices. In *USENIX Security*, 2022.

[16] Intel. Intel 64 and IA-32 Architectures Software Developer Manuals., 2023.

[17] Mohannad Ismail, Andrew Quach, Christopher Jelesnianski, Yeongjin Jang, and Changwoo Min. Tightly seal your sensitive pointers with PACTight. In *USENIX Security*, 2022.

[18] Jinghao Jia, Raj Sahu, Adam Oswald, Dan Williams, Michael V. Le, and Tianyin Xu. Kernel extension verification is untenable. In *Proceedings of the 19th Workshop on HOTOS*, 2023.

[19] Di Jin, Vaggelis Atlidakis, and Vasileios P. Kemerlis. EPF: Evil packet filter. In *USENIX ATC*, 2023.

[20] Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. kGuard: Lightweight kernel protection against Return-to-User attacks. In *USENIX Security*, 2012.

[21] Paul Kocher, Jann Horn, Anders Fogh, , Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *S&P*, 2019.

[22] Dmitry Kuznetsov and Adam Morrison. Privbox: Faster system calls through sandboxed privileged execution. In *USENIX ATC*, 2022.

[23] Donghyun Kwon, Jangseop Shin, Giyeol Kim, Byoungyoung Lee, Yeongpil Cho, and Yunheung Paek. uxom: efficient execute-only memory on arm cortex-m. In *USENIX Security*, 2019.

[24] Hans Liljestrand, Thomas Nyman, Lachlan J Gunn, Jan-Erik Ekberg, and N Asokan. Pacstack: an authenticated call stack. In *USENIX Security*, 2021.

[25] Hans Liljestrand, Thomas Nyman, Kui Wang, Carlos Chinea Perez, Jan-Erik Ekberg, and N. Asokan. Pac it up: Towards pointer integrity using arm pointer authentication. In *USENIX Security*, 2019.

[26] Soo Yee Lim, Xueyuan Han, and Thomas Pasquier. Unleashing unprivileged ebpf potential with dynamic sandboxing. *1st Workshop on eBPF and Kernel Extensions, ACM SIGCOMM*, 2023.

[27] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In *USENIX Security*, 2018.

[28] Hongyi Lu, Shuai Wang, Yechang Wu, Wanning He, and Fengwei Zhang. Moat: Towards safe bpf kernel extension, 2023.

[29] Jinsong Mao, Hailun Ding, Juan Zhai, and Shiqing Ma. Merlin: Multi-tier optimization of ebpf code for performance and compactness. In *ASPLOS*, 2024.

[30] Larry McVoy and Carl Staelin. Lmbench: Portable tools for performance analysis. In *USENIX ATC*, 1996.

[31] MITRE. BPF CVE list. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=BPF, 2024.

[32] Sujin Park, Diyu Zhou, Yuchen Qian, Irina Calciu, Taesoo Kim, and Sanidhya Kashyap. Application-informed kernel synchronization primitives. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, 2022.

[33] Joseph Ravichandran, Weon Taek Na, Jay Lang, and Mengjia Yan. Pacman: attacking arm pointer authentication with speculative execution. In *ISCA*, 2022.

[34] The Linux Foundation. XDP, eXpress Data Path. https://www.iovisor.org/technology/xdp, 2016.

[35] The Linux Foundation. BCC: BPF Compiler Collection. https://www.iovisor.org/technology/bcc, 2023.

[36] Jiali Xu, Mengyao Xie, Chenggang Wu, Yinqian Zhang, Qijing Li, Xuan Huang, Yuanming Lai, Yan Kang, Wei Wang, Qiang Wei, and Zhe Wang. Panic: Pan-assisted intra-process memory isolation on arm. In *CCS*, 2023.

[37] Yuhong Zhong, Haoyu Li, Yu Jian Wu, Ioannis Zarkadas, Jeffrey Tao, Evan Mesterhazy, Michael Makris, Junfeng Yang, Amy Tai, Ryan Stutsman, et al. Xrp:in-kernel storage functions with ebpf. In *OSDI*, 2022.

[38] Jie Zhou, Yufei Du, Zhuojia Shen, Lele Ma, John Criswell, and Robert J. Walls. Silhouette: efficient protected shadow stacks for embedded systems. In *USENIX Security*, 2020.

[39] Zhe Zhou, Yanxiang Bi, Junpeng Wan, Yangfan Zhou, and Zhou Li. Userspace bypass: Accelerating syscall-intensive applications. In *OSDI*, 2023.

## A The Description of Security Properties

We summarized the descriptions of the security properties in Table 8. The verifier tracks all the BPF objects' sizes and permissions, and the pointers' bounds point to them, thus whenever a BPF object is accessed, it forbids underflow/overflow and permission mismatch (lines 1/2/5). For BPF-accessible kernel objects, the verifier uses its whitelist, which records all readable/writable fields of each type of kernel object, ensuring legal fields with correct memory access width (lines 3/4/6).

When calling a helper function, it ensures each parameter matches the formal arguments in the signature (line 7). It maintains the status of each stack slot, thus pointers can be spilled into the stack and restored to the register. However, the BPF program cannot store pointers in other memory, because their later usage is not tracked, thus unable to prevent pointer leakage (line 8). Any operation that may leak the pointers indirectly (e.g., bitwise, compare) is not allowed (lines 9/10).

The stack memory and BPF registers may contain residual kernel information (e.g., pointers), it forbids operation that may lead to uninitialized access (lines 11/12/13).

It only considers Spectre V1 and V4 attacks, which are mitigated via different solutions. For example, Spectre V1 attack is mitigated by the *alu_sanitizer*, which adds mask instructions to prevent possible OOB speculative access. It also restricts stack access to simplify the Spectre V1 analysis

Table 8: Security properties related to the full-path analysis.

| | Security property | Description |
|---|---|---|
| 1 | BPF object OOB I | Accessing BPF objects disallow underflow or overflow. |
| 2 | BPF object OOB II | BPF stack access is checked to prevent exceeding 512B. |
| 3 | kernel object OOB I | Computation of pointers to kernel objects (e.g., context objects) is disallowed, and the immediate offset in memory access instruction is checked based on a whitelist to ensure the BPF program accesses legal fields. |
| 4 | kernel object OOB II | The whitelist is used to rewrite kernel object access based on the immediate offset in the instruction to ensure the BPF program accesses legal fields. |
| 5 | permission violation I | Read-only maps cannot be written to, and the same applies to packets for specific BPF program types. |
| 6 | permission violation II | Specific fields of kernel objects are read-only, ensured by the whitelist check. |
| 7 | type mismatch | Argument pointers must be matched to prevent BPF programs from using helper functions to arbitrarily access kernel objects. Pointers cannot be passed as scalars to prevent pointer leakage. |
| 8 | pointer leakage I | Pointers are disallowed as BPF function return value or for writing into memory, except for pointer spill due to maintained stack slot states. |
| 9 | pointer leakage II | Pointers are disallowed from performing certain calculations like bitwise or inter-pointer calculations. |
| 10 | offset leakage | Pointers cannot be compared with other pointers or non-zero scalars to prevent offset leakage. |
| 11 | uninit reg read | Registers must be initialized before use. |
| 12 | uninit stack read I | The BPF stack must be initialized before use. |
| 13 | uninit stack read II | BPF programs cannot write to the stack with variable offsets, as the full-path analysis needs to maintain the state of each stack slot. If the variable offset relates to an uninitialized slot, it cannot decide whether this write will initialize it or not. |
| 14 | Spectre V1 filter | BPF programs cannot read from the stack with variable offsets to simplify the Spectre V1 analysis. |
| 15 | Spectre V1 masking | Pointer arithmetic instructions with possible Spectre V1 attack is masked with the max range deducted by the full-path analysis. |
| 16 | Spectre V4 barrier | Store with possible Spectre V4 attacks are identified and instrumented with barrier instruction. |
| 17 | kernel stack crash I | The max depth of BPF call frame cannot exceed 32 to avoid kernel stack crash. |
| 18 | kernel stack crash II | The size of the combined BPF stack for each BPF function cannot exceed the upper limit (512B). |
| 19 | timeout | BPF program cannot contain dead loops. |
| 20 | deadlock | BPF program cannot cause deadlock. |

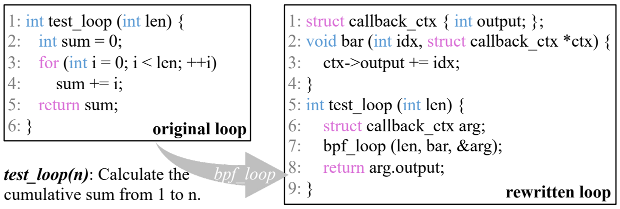
(lines 14/15). The barrier instruction mitigates Spectre V4 if a possible speculative store is detected (line 16).

The call frame of the BPF program cannot be too deep (e.g., less than 32) because it may crash the kernel stack (line 17). Similarly, the size of the combined BPF program stack of the deepest call frame cannot be too large (line 18).

The kernel disables the preemption when the BPF program executes, thus the BPF program cannot stuck (e.g., dead loop, deadlock) to keep the kernel functional (lines 19/20).

## B Complexity Issue Intensify with bpf_loop

Fig. 9 gives an example for the `bpf_loop()` helper, the original loop uses the `i < len` statement as the loop condition. Since the `len` variable may be unknown at the loading time, the full-path analysis has to speculate its range as large as possible (e.g., u32_max), thus failing to enumerate all program



```
int test_loop (int len) {
    int sum = 0;
    for (int i = 0; i < len; ++i)
        sum += i;
    return sum;
}
```

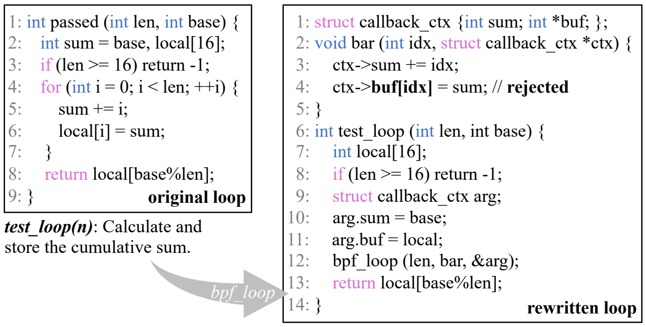
**original loop**

***test_loop(n)***: Calculate the cumulative sum from 1 to n.

```
struct callback_ctx { int output; };
void bar (int idx, struct callback_ctx *ctx) {
    ctx->output += idx;
}
int test_loop (int len) {
    struct callback_ctx arg;
    bpf_loop (len, bar, &arg);
    return arg.output;
}
```

**rewritten loop**

Fig. 9: The code example of `bpf_loop()`.



```
1: int passed (int len, int base) {
2:    int sum = base, local[16];
3:    if (len >= 16) return -1;
4:    for (int i = 0; i < len; ++i) {
5:        sum += i;
6:        local[i] = sum;
7:    }
8:    return local[base%len];
9: }
```

**original loop**

***test_loop(n)***: Calculate and store the cumulative sum.

```
 1: struct callback_ctx {int sum; int *buf; };
 2: void bar (int idx, struct callback_ctx *ctx) {
 3:    ctx->sum += idx;
 4:    ctx->buf[idx] = sum; // rejected
 5: }
 6: int test_loop (int len, int base) {
 7:    int local[16];
 8:    if (len >= 16) return -1;
 9:    struct callback_ctx arg;
10:    arg.sum = base;
11:    arg.buf = local;
12:    bpf_loop (len, bar, &arg);
13:    return local[base%len];
14: }
```

**rewritten loop**

Fig. 10: The negative example of `bpf_loop()`.

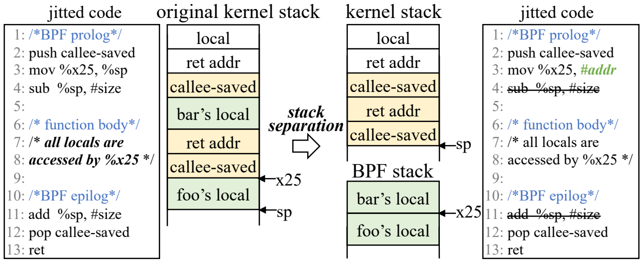
states, leading to program rejection. The bpf_loop() helper tries to solve this problem by encapsulating the loop body into a new function (`bar()` in the example), callback it using bpf_loop() with `len` as its call count and `arg` as its argument. The bpf_loop() will call the `bar()` for `len` times and pass the `index` from `0`. Inside the `bar()`, it stores the computed result and returns `0`, which means continue for the loop, while returning `1` means break the loop.

While this helper aims to address the complexity issue, it could worsen the problem. As the Fig. 10 shows, the original function can pass the verification but fails after being rewritten by the `bpf_loop()`. This is due to the inability to pass states between the loop body and condition. For example, in the original function, the `i` variable is always less than `len` (which is less than 16), thus the memory access instruction in line 6 is safe. After the loop body becomes an independent function, which is verified independently, the full-path analysis cannot ensure the range of `idx`, thus rejecting the BPF program for possible OOB access. Besides, this callback style function restricts pointer passing between function calls, thus the loop body cannot use any self-defined pointers (`buf` in line 4), which is another limitation.

## C The Details of Compartmentalization

### C.1 The BPF Stack Handling

The BPF stack is on top of the current kernel stack. Besides, the return address and callee-saved registers (*metadata*) are also stored in the kernel stack and are only accessed by `%sp`.



```
 1: /*BPF prolog*/
 2: push callee-saved
 3: mov %x25, %sp
 4: sub  %sp, #size
 5:
 6: /* function body*/
 7: /* all locals are
 8: accessed by %x25 */
 9:
10: /*BPF epilog*/
11: add  %sp, #size
12: pop callee-saved
13: ret
```

```
 1: /*BPF prolog*/
 2: push callee-saved
 3: mov %x25, #addr
 4: ~~sub  %sp, #size~~
 5:
 6: /* function body*/
 7: /* all locals are
 8: accessed by %x25 */
 9:
10: /*BPF epilog*/
11: ~~add  %sp, #size~~
12: pop callee-saved
13: ret
```

Fig. 11: The stack separation process. The kernel calls the `bar()` in BPF, and the `bar()` calls the `foo()` in BPF.

As shown in Fig. 11, HIVE moves the local variables into BPF space by replacing the initialization of `%x25` in the prologue (`mov %x25, %sp`) with another instruction (`mov %x25, #addr`), and leaving the *metadata* on the kernel stack. The stack adjustment instructions (lines 4/11) are optimized away. HIVE instruments instructions to adjust the `%x25` to manage the BPF stack when entering the BPF program and calling BPF functions. Without full-path analysis, HIVE does not know the size of each stack frame, so it uses the upper limit (512B) allowed in eBPF as the fixed size of each frame. The size of the BPF stack is set to be 8MB by default, HIVE only allocates the bottom two pages at load time, and the remaining pages are allocated lazily via the page fault exception.

Since the control flow safety is guaranteed (forward control flow is ensured by the first two stages of the verifier, and backward control flow is ensured by return address isolation), instructions that operate `%sp` are emitted/instrumented by the JIT compiler and cannot be abused by attackers. The advantage of this design is that there is no need to switch the physical stack pointer `%sp` when the kernel function and the BPF function call each other. During the BPF program's execution, the interrupt handlers and the helpers can be called directly as they both use the kernel stack.

## C.2 The Maps Separation

There are 33 distinct types of maps in the current eBPF with mainly two types of implementations: array maps and hash maps. The shadowing process is straightforward for array maps since they are stored continuously and of a fixed size. eBPF provides map properties to page-align the values, which helps HIVE to only shadow the data area of the maps.

However, the hash maps are stored non-contiguously, preventing HIVE from shadowing the hash maps directly into the BPF space. To this end, we slightly adjust the structure of the maps, separating all values from the maps and leaving a pointer to point to the new location of the values. Fig. 12 gives the separation of the hash_map structure, all values are separated and stored in the BPF space, while the metadata remains in the kernel space. The change in the map structure does not affect the normal functionality of BPF programs.

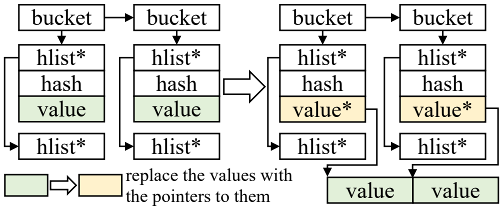


Fig. 12: The hash map separation.

## D Miscellaneous Helpers Rewriting

The 11 helpers that need rewriting are introduced as follows:

- *bpf_spin_[lock|unlock].* The full-path analysis needs to ensure no spinlock is acquired in the current program state when calling the `bpf_spin_lock()`, and the lock must be released before the BPF program exits. To this end, HIVE handles these helpers by adding additional record-and-check logic in them. For example, HIVE maintains a per-program spinlock table for the `bpf_spin_lock()`, each time this helper is called, HIVE checks if the current BPF program has acquired a spinlock already, if true, the BPF program will be uninstalled. Otherwise, HIVE updates the spinlock table. This table also ensures the BPF program will release the spinlock through the `bpf_spin_unlock()` helper before the BPF program exits.
- *bpf_timer_[init|set_callback|start|cancel].* eBPF provides callback timer functions via these 4 functions, and the full-path analysis ensures all the timers are initialized by use, as well as their integrity. Since these functions all require the state of the timers, HIVE maintains a timer table for them, and updates the table based on their functionalities (e.g., insert an entry in the table when the texttttbpf_time_init is called). Once a timer is corrupted by the BPF program, HIVE can detect that in the helper by the timer table.
- *bpf_dynptr_[from_mem|read|write|data].* A dynptr is a pointer that stores metadata alongside the address. The full-path analysis ensures it is read-only to the BPF program and can only be used through helpers. To this end, HIVE maintains a dynptr table for them. Each time a dynptr is generated, it will be recorded into the table, and HIVE checks the table when the dynptr is used. Different from the full-path analysis, the BPF program can corrupt the dynptr, but it will be identified once the dynptr is used.
- *bpf_kptr_xchg.* Unlike other PTR_TO_BTF_ID helpers, the btf_id in this helper is determined dynamically by the full-path analysis, which ensures the corresponding argument has BTF information in the maps when calling this helper. Since HIVE cannot determine which path contains which map with BTF information, it needs to dynamically find the map by the pointer value and ensure the corresponding map contains the BTF information.