SoK: Challenges and Paths Toward Memory Safety for eBPF

Kaiming Huang
The Pennsylvania State University
kzh529@psu.edu

Mathias Payer
École Polytechnique Fédérale de Lausanne
mathias.payer@nebelwelt.net

Zhiyun Qian
University of California, Riverside
zhiyunq@cs.ucr.edu

Jack Sampson
The Pennsylvania State University
jms1257@psu.edu

Gang Tan
The Pennsylvania State University
gxt29@psu.edu

Trent Jaeger
University of California, Riverside
trentj@ucr.edu

***Abstract*—The extended Berkeley Packet Filter (eBPF) subsystem in Linux enables the extension of kernel functionality without modifying kernel code. In addition to its use in networking, eBPF provides the flexibility to perform tracing, add security checks, etc. To ensure that eBPF does not enable attackers to compromise the kernel, eBPF includes a verifier to validate every eBPF program before its execution, which includes checks that aim to prevent eBPF programs from modifying kernel memory due to memory errors. However, numerous vulnerabilities have been identified in the eBPF subsystem, including the verifier itself, which greatly violate expectations, leading to concerns about the threats of memory safety brought by eBPF. This paper presents the first systematic analysis of the memory safety risks inherent in the eBPF ecosystem, focusing on the challenges faced by the limitations of the eBPF verifier and current kernel defenses. We then evaluate proposed research mitigation strategies that apply isolation techniques, runtime checks, and static validation, highlighting their contributions and gaps. Our study finds that only 1.62-3.74% (37-85) of the memory operations in public eBPF programs cannot be proven memory safe comprehensively, motivating actionable insights towards enforcing comprehensive memory safety while accounting for performance and compatibility.**

1. Introduction

The extended Berkeley Packet Filter (**eBPF**) [1], [2] enables the dynamic execution of user-defined programs in kernel space. Originally developed to enhance network packet filtering [3], [4], [5], [6], [7], eBPF now supports a broad range of applications beyond networking, including performance monitoring [8], [9], [10], [11], [12], security enforcement [13], [14], [15], [16], [17], and system tracing [18], [19], [20], [21], [22]. eBPF defines a bytecode format that can be either interpreted or JIT-compiled, enabling verification of both its semantics and security properties. eBPF has reshaped how developers monitor and control the system by enabling direct interaction with kernel functions and data, making it possible to build high-performance, customizable programs without constantly modifying the kernel. This unique capability allows eBPF programs to operate with exceptional efficiency, bypassing traditional overheads such as context switching, which are critical in high-throughput, latency-sensitive environments. However, these powerful capabilities come with significant security challenges, such as cross-container attacks [23], transient execution attacks [24], [25], and particularly, memory safety within kernel space.

The privileged execution of eBPF programs in the kernel context, though performance-oriented, opens the system to potential memory error exploitations that can compromise the entire kernel [26]. In fact, the Linux eBPF has been blamed as a *new privilege escalation technique* in 2022 [27], and has been used as an effective measure to enhance attacker's capabilities and expanding the exploitability of kernel vulnerabilities [28], [29]. Memory safety is essential in eBPF's architecture, as flaws here can lead to severe security issues, including arbitrary code execution, privilege escalation, data corruption, and denial of service. To mitigate these risks, the Linux kernel integrates the eBPF verifier [30], a static analysis tool designed to assess eBPF programs for safety violations before allowing their execution. The verifier enforces various security checks, such as validating memory access bounds and ensuring proper pointer usage. Despite these precautions, the verifier has shown limitations in comprehensively blocking unsafe operations [31]. These gaps are especially concerning given the expanding attack surface presented by eBPF's deep integration into the kernel.

Recent years have seen a surge (see Section 3) in reports of eBPF-related vulnerabilities [32], highlighting the growing demand for more resilient memory safety mechanisms. Notably, many malicious eBPF programs can bypass verifier checks intentionally by leveraging complex program structures or exploiting specific verification weaknesses, resulting in vulnerabilities that can lead to critical kernel corruption. Over the years, kernel maintainers keep adding checks into the eBPF verifier to prevent such circumvention and to support new features added to the eBPF subsystem. In Linux kernel version v6.11, the verifier contains more than 22k lines of code, which is 11x more than its initial version v3.18. For example, 93 lines of code were added into the

verifier in 2024 to prevent an integer overflow [33], and 595 lines of code were added into the verifier in 2023 for supporting verification on open-coded iterator loops [34]. Unfortunately, even as the verifier is refined, sophisticated attacks continue to find ways to evade, suggesting that memory safety in eBPF is far from ensured. In 2024 alone, over 100 bugs related to memory errors were discovered in the eBPF subsystem, with 45 of them still unaddressed [35]. Furthermore, a substantial number of CVEs regarding memory errors have been linked to eBPF since 2023, underscoring the need for improved memory safety solutions.

Beyond the verifier, other components in the eBPF ecosystem pose additional security risks. For instance, eBPF helper functions are APIs provided by the kernel that allow eBPF programs to perform specific operations, such as accessing packet data or managing eBPF maps—data structures that facilitate communication between user space and kernel space. These helper functions, however, generally lack internal safety checks, relying instead on the verifier to enforce safe usage. This reliance creates a trust chain that can be easily broken if an unsafe eBPF program slips past the verifier. Consequently, flawed or malicious eBPF programs can exploit unchecked helper functions to access or corrupt kernel memory, bypassing isolation mechanisms designed to protect the kernel. The risk is compounded by the complexity of interactions within the eBPF subsystem, where unchecked data in maps or pointers manipulated by helpers can lead to memory errors that escape the verifier.

To address these challenges, various defense strategies have been proposed. Isolation techniques, including both software-based fault isolation [36], [37] and hardware-assisted mechanisms [38], [39], aim to contain eBPF programs within safe memory boundaries. While isolation provides an essential layer of defense, it is often insufficient to fully address the spectrum of memory vulnerabilities due to indirect kernel interactions (e.g., cross-boundary interface vulnerabilities (CIVs) [40], [41] through eBPF helpers). Static and dynamic verification enhancements have also been explored [42], [43], [44], [45], [46], [47], [48], [49], such as refining the verifier's capabilities to track dependencies and reject unsafe memory operations. Efforts to migrate eBPF towards memory-safe languages, such as Rust [31], offer the potential to eliminate certain classes of memory errors at the source level. Nevertheless, each of these approaches presents trade-offs, such as performance overhead, limited coverage, or compatibility challenges.

Based on the facts discussed above, we assert that while awareness is growing around the eBPF subsystem's threat to kernel memory safety[1], *the specific weaknesses and flaws within each component, across the workflow, and in existing defenses remain alarmingly unaddressed*. There is an urgent need to systematically document not only the current threats eBPF poses to kernel memory, but also the limitations of existing defenses. By +identifying and summarizing these weaknesses, this work aims to further point out essential future directions for effective and comprehensive memory safety solutions within the eBPF framework, advancing toward a more secure foundation for the Linux kernel amid mounting and potentially catastrophic vulnerabilities.

In this paper, we start with a summary of the workflow and trust model of the eBPF subsystem (Section 2), revealing that the trust model relies critically on the eBPF verifier. However, we then show that the eBPF verifier has proven unreliable, with numerous vulnerabilities allowing unsafe programs to bypass its checks, making it a significant source of kernel memory safety issues (Section 3). To address the sources of these vulnerabilities, we provide a detailed study that classifies the weaknesses of the eBPF verifier into three categories using concrete CVEs (Section 4), reflecting that the checks are unsound, incomplete, and have a limited scope that prevents comprehensive validation of memory safety. The Linux kernel includes both eBPF-specific and general defenses to mitigate issues arising from an unreliable verifier, but we find that eBPF-specific defenses are hindered by optional configurations and limited enforcement of privilege restrictions, while general defenses lack the coverage needed to fully prevent eBPF-based attacks, leaving the kernel vulnerable (Section 5).

Researchers have recognizes these shortcomings in the eBPF verifier and current kernel defenses and proposed a variety of defenses aiming to prevent memory errors and/or their exploitation. This paper systematically reviews these research advances to enhance eBPF memory safety, including improvements to static verification, isolation, runtime checks, fuzzing, and migration to memory-safe languages (Section 6). While each approach offers specific security benefits, they also present limitations: static verification struggles with validating complex runtime interactions, isolation contains memory access but can be bypassed indirectly, and runtime checks improve safety but are performance intensive. Fuzzing helps uncover vulnerabilities yet lacks full coverage, and language migration introduces compatibility issues and requires significant changes to the kernel. Effective security for eBPF requires balancing these strategies to meet both memory safety requirements and high-performance/flexibility demands.

We identify the key weaknesses in the current eBPF security model, including limited privilege restrictions, incomplete checks by the verifier, lack of safeguards in eBPF helpers, and coarse-grained isolation techniques. Additionally, runtime checks prove too costly for high-performance eBPF use cases, exposing vulnerabilities and potential attack vectors that could exploit these gaps (Section 7). We evaluate public and malicious eBPF programs to determine how many unsafe memory operations that need attention and propose potential directions for enforcing memory safety comprehenisvely (Section 8). We aim to provide actionable research directions to prevent the exploitation of memory errors, enabling safe deployment within the Linux kernel. We make the following contributions in this paper:

- ***What can go wrong***: We conduct the first systematic study of the memory safety threats that the eBPF subsystem poses to the Linux Kernel.

1. We will only focus on memory safety of eBPF, other security concerns (e.g., concurrency bugs and side channels) are excluded from the scope.

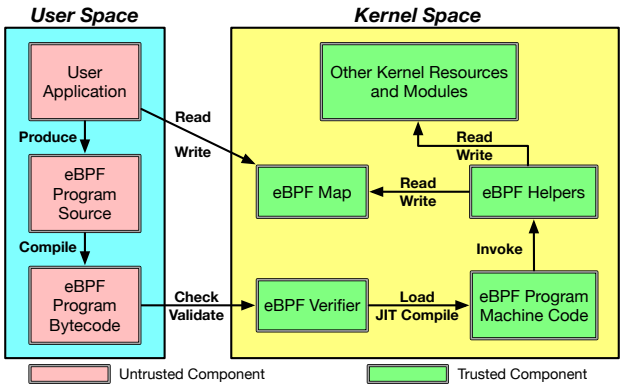

Figure 1: Overview of the eBPF workflow and its trust model.

- ***How does it go wrong***: We conduct the first systematic study of the weaknesses in the workflow of the current eBPF subsystem that introduces memory safety threats.
- ***Has the issue been resolved***: We conduct the first comprehensive study of currently deployed and research defenses by classes, highlighting their contributions while pinpointing individual shortcomings.
- ***How to fix it***: We provide guidance for potential future research directions for enhancing eBPF memory safety.

2. eBPF Workflow and Trust Model

The extended Berkeley Packet Filter (eBPF [50]) framework allows user-defined programs to execute within the Linux kernel in a secure, sandboxed environment, but eBPF programs stand apart from regular programs. These programs are subject to stricter regulations due to their direct interaction with the kernel, and are highly performant through the JIT compilation. Originally created for network packet filtering, eBPF now supports a broad range of critical tasks, including tracing, security, and performance monitoring.

In the current Linux kernel security model, the trust hierarchy establishes which components are trusted and which are not, as shown in Figure 1. Initially, the *eBPF programs* are untrusted since they originate from untrusted *user-space applications* and are installed in the context of untrusted users. These programs are written by users, compiled into bytecode, and vetted by the *eBPF verifier* before they can execute within the kernel. The verifier is a critical, trusted component that performs static analysis on eBPF programs, checking for memory safety, pointer integrity, stack/register usage, and enforcing limits on loops and instruction counts to prevent unsafe operations. Only after passing these checks can an eBPF program be loaded, JIT compiled into machine code, then executed in the kernel.

Once verified and loaded, eBPF programs are trusted to perform only benign memory accesses. eBPF programs rely on trusted *eBPF helper* functions, which are kernel-provided APIs that offer trusted access to kernel operations like packet parsing, map management, and limited memory access. Using eBPF helpers, eBPF programs may access kernel data structures shared with the eBPF subsystem. Examples of such data structures include eBPF maps and other kernel resources, such as ring buffers, Perf buffers, and tracepoints. For example, *eBPF maps* serve as intermediaries for data exchange between user and kernel memory, providing data persistence across program executions. User-space applications can access the eBPF map through system calls (i.e., `BPF()`). Beyond the eBPF subsystem, eBPF programs can indirectly affect *other kernel resources and modules* through eBPF helpers e.g., `bpf_skb_load_bytes`. These resources, again, depend on the verifier and helpers to enforce memory safety and access control to prevent unauthorized access.

Notably, the chain of trust from verified eBPF programs to other kernel resources, is only as strong as the eBPF verifier. If the eBPF verifier fails to identify unsafe behaviors or vulnerabilities in an eBPF program, no further checks on ensuring memory safety of eBPF program occur at runtime. Specifically, eBPF helper functions, though trusted, rely on the eBPF verifier to ensure that eBPF programs use them safely; they do not perform additional validation of memory access or pointer safety. Consequently, if a malicious or flawed eBPF program bypasses the verifier, it could misuse helpers to manipulate kernel memory, such as forging a pointer or modifying map data in a way that grants unauthorized access to broader kernel memory, potentially leading to kernel memory corruption or privilege escalation. Thus, any weaknesses in the verifier can compromise the entire trust chain, exposing the kernel to significant security risks from untrusted eBPF programs (see examples in Section 4).

TA 1. [a]**The eBPF trust model relies critically on the eBPF verifier to enforce memory safety; If the verifier fails, untrusted or malicious eBPF programs may corrupt kernel memory (e.g., misuse eBPF helpers).**

a. We identify key observations as "Take-Aways". TAs with key insights are highlighted in bold, other TAs are summaries of knowledge.

3. Memory Safety Issues in the eBPF Verifier

The eBPF verifier serves as the only defense in the Linux kernel to analyze and reject unsafe eBPF programs. Unfortunately, it also has become a significant source of bugs. A substantial amount of CVEs have been reported for bugs identified in the eBPF verifier, including 46 from year 2024 after kernel maintainers became reluctant to assign CVEs to kernel bugs [51], [52]. In fact, the verifier serves as the most vulnerable components in terms of the number of CVEs discovered in the eBPF subsystem [53]. These bugs enable attackers to bypass security checks, and the eBPF subsystem has been becoming a hotbed for introducing memory safety vulnerabilities into the kernel. To date, Syzbot reported a total of 325 bugs that are caused by memory errors [32] since eBPF was added into the Linux kernel in version 4.4 [54] (Jan 2016), making the eBPF subsystem the fourth-ranked subsystem in the Linux kernel in terms of the number of bugs. More importantly, more than half (169) were identified since 2023, and 45 of them remain unfixed [35].

Specifically, all aforementioned bugs were triggered by eBPF programs generated by the kernel fuzzer, SyzKaller [55], indicating that all of them can bypass the eBPF verifier checks and get loaded into the kernel. The reported bugs, if exploited by an attacker, may cause serious consequences to the kernel, such as privilege escalation, arbitrary code execution, and DoS. The sharp increase in identified bugs since 2023, with many remaining unresolved, highlights the weakness of the eBPF verifier to identify unsafe operations that may lead to memory errors, resulting in the ongoing and growing challenge of ensuring that eBPF programs satisfy memory safety.

TA 2. The eBPF verifier, intended as the Linux kernel's primary defense to reject unsafe eBPF programs, has failed to reject programs that may violate memory safety, becoming a major source of vulnerabilities.

4. Attack Flawed Verification

In this section, we will provide a detailed analysis of the weaknesses of the eBPF verifier in identifying memory safety threats. We examined the complete set of memory safety vulnerabilities discovered in 2024 that have concrete PoCs and reproducers, including 85 Syzbot generated reports and 17 CVEs. Based on the investigation, we found that existing checks in the eBPF verifier have shown weaknesses in three aspects (1) *Unsound* - checks are implemented in an unsound way that can be bypassed, such as [56], [57], [58], [59], [60]; (2) *Incomplete* - necessary checks needed to achieve full memory safety are missing, such as [61], [62], [63], [64], [65]; (3) *Limited scope* - checks are only conducted towards eBPF bytecode, ignoring other components in the eBPF subsystem, such as [66], [67], [68], [69], [70]. We use real-world vulnerabilities as examples to concretly demonstrate how these weaknesses allow eBPF programs to violate memory safety. All examples are shown to bypass all the currently deployed kernel defenses listed in Table 1 and damage the kernel.

4.1. Bypass Unsound Checks

Declared checks in the eBPF verifier can be unsound, enabling an attacker to bypass existing checks. For example, one of the eBPF verifier's functionalities is path pruning [71], which skips redundant execution paths to improve verification performance. However, accurate path pruning depends on precise tracking of dependencies among registers. If the verifier fails to track a dependency between registers accurately, unsafe paths may be mistakenly pruned as safe. CVE-2023-2163 [72] is one such case. This issue allows attackers to exploit the verifier's path-pruning mechanism to bypass memory safety checks, potentially leading to privilege escalation and arbitrary memory write operations.

Specifically, when an eBPF program executes pointer arithmetic operations, all registers involved in such operations should be tracked. For instance, if register `r1` undergoes pointer arithmetic involving `r2` (e.g., `r1=r1+r2`), the verifier should recognize that `r1`'s value depends on `r2`. This dependency requires the verifier to explore all paths involving `r1` under the influence of `r2`. However, in CVE-2023-2163, the verifier mistakenly ignores `r2` as a dependency of `r1`, leading it to prune paths prematurely. This premature pruning means that when `r1` points to memory locations (base) influenced by `r2` (offset), the verifier incorrectly assumes it is always within safe bounds, leading to out-of-bounds access. By crafting eBPF programs with pointer operations involving `r1` and `r2`, attackers can cause `r1` to point outside its allocated boundary.

```
1  SEC("socket")
2  int exploit_prog(struct sk_buff *skb){
3      int *data; //address stored in r1
4      data = bpf_ringbuf_reserve(&ringbuf, 64, 0);
5      if (!data)  return -1;
6      // Exploiting path pruning by passing an out-of-
       bounds pointer in r1
7      // Here, r2 is the offset that exceeds bounds
8      bpf_ringbuf_submit(data + 128, 0);
9      // r1 = data + 128, r2 = 128
10     return 0;
11 }
```

Listing 1: Example PoC for CVE-2023-2163.

To exploit this vulnerability, the attacker must craft a malicious eBPF program that tricks the verifier into disregarding the dependency between `r1` and `r2`, thereby prematurely pruning the verification of the path corresponding to `r1 = r1 + r2`. The PoC shown in Listing 1 demonstrates how this vulnerability can be easily exploited. `data` representing a pointer to the reserved memory block (size 64), is stored in r1. For simplicity, the comment at line 6 represents the steps that the attacker deceives the verifier, the details are illustrated in the CVE writeup [73], [74]. The `bpf_ringbuf_submit` function is then called with an out-of-bounds pointer (`data+128`). Here, 128 (stored in `r2`) acts as an offset applied to `r1`, shifting it beyond the bound. Because of the missing dependency tracking between `r1` and `r2` (triggered by attacker), the verifier assumes the memory access is within bounds and prunes this path (i.e., the call at line 8) as if it were safe, allowing the out-of-bounds access to occur unchecked. This flaw grants the attacker the arbitrary memory accesses in kernel memory.

TA 3. Optimizations that improve verifcation performance of the the eBPF verifier can be unsound. Flaws in removing checks by optimizations allow attackers to instantiate programs that violate memory safety.

4.2. Exploit Gaps in Incomplete Checks

Checks performed by the verifier are insufficient to guarantee comprehensive memory safety, i.e., checks that should be included are missing. CVE-2021-4204 [75] is a vulnerability found in the eBPF subsystem that stems from a lack of proper argument validation for helper functions, which permits out-of-bounds memory accesses that can lead to privilege escalation. The eBPF verifier, which should

report such an improper argument specification, let the program pass the verification, resulting in the vulnerability.

One of the responsibilities of the eBPF verifier is validating that the use of helper functions obeys the required calling conventions [30]. Each helper function must follow its predefined structure provided by struct `bpf_func_proto`, specifying argument types and return values. The verifier checks that these arguments are valid before the program can be loaded. For example, `bpf_strtol()` accepts a pointer to memory and the size of the memory block as its arguments. The verifier uses the size argument to validate that the size of memory block is correct, preventing out-of-bounds memory access, as shown below in Listing 2.

```
const struct bpf_func_proto bpf_strtol_proto = {
    .func       = bpf_strtol,
    .gpl_only   = false,
    .ret_type   = RET_INTEGER,
    .arg1_type  = ARG_PTR_TO_MEM, // Pointer to string
    .arg2_type  = ARG_CONST_SIZE, // Size of the string
    .arg3_type  = ARG_ANYTHING,   // Additional argument
    .arg4_type  = ARG_PTR_TO_LONG,// Pointer to result
};
```

Listing 2: Signature of bpf_strtol()

Argument 1 (`ARG_PTR_TO_MEM`) is a pointer to a memory block. Argument 2 (`ARG_CONST_SIZE`) specifies the size of the memory block. The verifier ensures that the pointer passed in the first argument is valid and that the length specified in the second argument correctly matches the allocated memory. If the size is incorrect, the verifier will reject the program. For example, if `bpf_strtol("hello",100,0,&res)` is attempted, the eBPF verifier rejects this eBPF program because the provided size (100) exceeds the length of the string "`hello`".

The cause of CVE-2021-4204 is that the definition of helper functions `bpf_ringbuf_submit()` and `bpf_ringbuf_discard()` do not have the size argument of memory block as part of their expected calling convention. These functions take a pointer to allocated memory (`PTR_TO_ALLOC_MEM`) as the first argument, but they lack a size argument for the verifier to validate the memory block's size, as shown in Listing 3.

```
const struct bpf_func_proto bpf_ringbuf_submit_proto={
    .func        = bpf_ringbuf_submit,
    .ret_type    = RET_VOID,
    .arg1_type   = ARG_PTR_TO_ALLOC_MEM,
    .arg2_type   = ARG_ANYTHING,};

const struct bpf_func_proto bpf_ringbuf_discard_proto={
    .func        = bpf_ringbuf_discard,
    .ret_type    = RET_VOID,
    .arg1_type   = ARG_PTR_TO_ALLOC_MEM,
    .arg2_type   = ARG_ANYTHING,};
```

Listing 3: Flawed Helper Function Signature Definition.

In Listing 3, both of the function signatures define the first argument as a pointer to allocated memory (`PTR_TO_ALLOC_MEM`). However, the verifier will not perform size validation for the memory block, since there is no parameter in the function signature that specify the legitimate size for the eBPF verifier to check. In this case, the verifier simply skips such checks, rather than rejecting the program or at least issuing a warning. The lack of proper size validation allows a user to pass out-of-bounds pointers easily by using input of arbitrary length that is referenced by the first argument of the function signature, leads to memory corruption in the kernel memory.

```
SEC("xdp_prog")
int my_prog(struct xdp_md *ctx) {
    void *sample;
    // Reserve a memory block in the ring buffer
    sample = bpf_ringbuf_reserve(&ringbuf, 64, 0);
    if (!sample)
        return XDP_ABORTED;
    // Perform some operation on the buffer
    bpf_probe_read(sample, 64, "data");
    // Exploit: Call with an out-of-bounds pointer
    bpf_ringbuf_submit(sample + 128, 0);
    // OOB pointer, bypasses size check
    return XDP_PASS;
}
```

Listing 4: Example PoC of the Vulnerability.

To exploit this vulnerability, the attacker just needs to reserve memory in the ring buffer[2], then write data into the allocated memory block, and finally invoke `bpf_ringbuf_submit()` with an out-of-bounds pointer as parameter. As shown in the example PoC in Listing 4, the malicious eBPF program reserves 64 bytes of memory in the ring buffer, then the flawed `bpf_ringbuf_submit()` helper function is called with an out-of-bounds pointer `sample+128` at line 11, bypassing the intended memory bounds. Since there is no size check for the pointer passed to `bpf_ringbuf_submit()` the attacker can manipulate the internal logic of the ring buffer to execute arbitrary code in kernel space, leading to privilege escalation.

TA 4. The eBPF verifier trusts eBPF helpers unconditionally, checks that are required for comprehensive memory safety are missing in the verifier.

4.3. Exploiting Limited Protection Scope

The eBPF trust model has fundamental weaknesses. Specifically, the eBPF verifier places implicit trust in eBPF helper functions, assuming they are always used safely by programmers. Thus, verifier checks are only limited to the eBPF bytecode without going into other parts of the subsystem (e.g., eBPF helpers).

For example, the eBPF verifier's temporal memory safety checks are limited to eBPF bytecode [76], with little detail on how or which cases are covered. The only documented checks apply strictly to Kfunc parameters [77], and other sources provide no clear detection mechanisms [78]. Thus, this not only reiterates that the implemented checks are incomplete (Section 4.2), but also emphasizes that memory errors that are not presented in the bytecode (e.g., in the helpers) still exist in the wild. For instance, a program may result in use-before-initialization using `dev_map_lookup_elem()`, as mentioned in a Syzbot bug report [79]. eBPF helpers generally ignore memory safety checks and trust that the eBPF program uses them properly, as the source code of `dev_map_lookup_elem()` shows in Listing 5.

2. Ring buffer is another kind of intermediate data structure between kernel, eBPF program, and user space, similar to eBPF map

`dev_map_lookup_elem()` retrieves an entry from a `dev_map` using the provided key without verifying whether the entry has been initialized. The function relies solely on `rcu_dereference_check()` to ensure safe concurrent access but does not validate that `obj` is initialized. This leaves opportunities for attacks, for example, the C reproducer in the bug report [80], as shown in Listing 6.

```
static void *__dev_map_lookup_elem(
                struct bpf_map *map, u32 key){
    struct bpf_dtab *dtab =
                container_of(map, struct bpf_dtab, map);
    struct bpf_dtab_netdev *obj;
    if (key >= map->max_entries)
       return NULL;
    obj = rcu_dereference_check(dtab->netdev_map[key],
              rcu_read_lock_bh_held());
    return obj;
}
```

Listing 5: Source Code of dev_map_lookup_elem()

```
SEC("classifier")
int example_prog(struct __sk_buff *skb) {
    int index = 0; // Key for accessing dev_map
    int *dev_ifindex;
    // Use dev_map_lookup_elem to retrieve the interface
    dev_ifindex = dev_map_lookup_elem(&dev_map, &index);
    if (!dev_ifindex) {
        return TC_ACT_SHOT; // Drop packet if fails
    }
    // Uninitialized memory access
    *dev_ifindex += 1; // KMSAN uninit warning
    // Final decision to accept or drop the packet
    return TC_ACT_OK;
}
```

Listing 6: Reproducer eBPF Program of Use Before Initialization in Syzbot Report [80]

The issue arises when `dev_map_lookup_elem` is used to retrieve an entry from the `dev_map`. If this entry is uninitialized, the `dev` pointer returned may point to uninitialized memory. Subsequently, modifying `ifindex` without prior initialization leads to undefined behavior, as this memory location may contain arbitrary data. This causes a temporal memory safety violation, which KMSAN detect by flagging accesses to uninitialized memory. Since `dev_map_lookup_elem` does not ensure initialization before any use, the responsibility falls on the eBPF program to verify or initialize entries, creating an exploitable gap since the existing verifier does not verify it as well.

TA 5. Checks of the verifier are limited to the eBPF bytecode, but they should be extended to all components in the subsystem.

5. Kernel Mitigation Circumvention

It has long been known that the eBPF verifier may have bugs that can cause it to execute malicious eBPF programs that should have been rejected [38], [81]. In addition to the eBPF verifier, the Linux kernel has deployed three other eBPF-specific defenses designed to prevent attacks by malicious eBPF programs, as listed in Table 1, namely the CAP_BPF capability [82], BPF LSM [83], and BTF/CO-RE [84]. These defenses are not required, and sometimes disabled intentionally (e.g., BPF LSM is often disabled for performance reasons), so malicious eBPF programs may bypass these optional protections when they are not enabled. The Linux kernel recently introduced CAP_BPF[3] privilege [82], [86] in Linux 5.8 (Aug 2020) to block unprivileged users from attaching eBPF programs [87]. However, this CAP_BPF restriction is ***not a hard constraint***, meaning that unprivileged users can still opt out of this restriction and attach their eBPF programs into the kernel [88]. In fact, requiring privileges to run eBPF programs significantly limits their flexibility and portability [36]. Many vendors rely on eBPF's unprivileged execution model for ease of use and deployment. For instance, Cilium [89], a popular eBPF-based networking and security platform, takes advantage of unprivileged eBPF programs to ensure seamless integration across multiple environments, including cloud-based systems. As a result, vendors continue to operate with unprivileged eBPF [90]. This tradeoff reflects the challenges in balancing security and usability in real-world applications of eBPF, highlighting that privilege separation is not a panacea. Finally, even enforcing CAP_BPF does not prevent attackers from exploiting memory errors in (mistakenly) verified eBPF programs to gain unauthorized kernel access [72] (e.g., root privilege), such attacks [91], [92] are shown to not be prevented by the defenses in Table 1.

In addition to specific eBPF defenses, some general-purpose kernel defenses (listed at the bottom of Table 1) that aim to prevent kernel memory error exploitation provide additional protection of the memory safety of eBPF. However, these defenses face two primary limitations. First, they are *incomplete* in scope: While they address certain attack vectors, they do not cover all the unique vulnerabilities introduced by eBPF, leaving some attack surfaces exposed. Second, these defenses can be *circumvented* by advanced exploits, making it possible for attackers to exploit kernel memory despite the presence of these mechanisms. Even in combination, these general defenses are insufficient to fully secure the kernel against the range of threats posed by malicious eBPF programs.

In summary, while all defenses listed in Table 1 collectively enhance security, they cannot fully address the memory safety threats created by erroneously "verified" eBPF programs. New defenses are needed to address these threats directly and comprehensively.

TA 6. The Linux kernel's eBPF-specific defenses are limited by optional settings and left room for attacks with limited privilege, while general defenses fail to fully block eBPF-based attacks.

6. Summary of State-of-the-art Solutions

eBPF has become a crucial vehicle for extending kernel functionality. Despite its versatility, this extensibility introduces a significant threat to memory safety. This section

3. Directly assigning root privilege is undesirable, equivalent to granting an arbitrary user-level program root permission, so defenses to split BPF permissions from root privileges has become a trend [85].

Category	Kernel Defensive Features	Description
Required Defense	eBPF Verifier	Validates security of eBPF programs.
Optional Defense	Capability CAP_BPF	Permits only privileged users to attach eBPF programs.
	BPF LSM (Linux Security Modules)	Enforces access control over eBPF programs
	BPF Type Format (BTF) and CO-RE	Validates data type and version compatibility.
General Defense	CFI and Execute-Only Memory (XOM)	Prevents control flow hijacking and code reuse attacks.
	Memory Tagging	Prevents pointers from being tampered and forged.
	Shadow Stacks	Protects return addresses.
	kASAN	Detects memory errors at runtime.
	kASLR	Randomizes memory layout.
	SMAP and SMEP	Prevents unauthorized user-space memory access in kernel mode.

TABLE 1: Summary of kernel defenses on eBPF: classified as Required/Optional for eBPF, or General for kernel.

Fuzzer	Description	eBPF-Specific
Syzkaller [55] libFuzzer [93] AFL [94]	General-purpose fuzzers for identifying memory errors, need specific configuration for eBPF fuzzing.	✗
Buzzer [95] BRF [96] BpfChecker [97] LKL Fuzzer [53]	eBPF-specific fuzzers that are effective in generating eBPF programs that pass verifier checks using different techniques.	✓

TABLE 2: Summary of Fuzzing Tools for eBPF

reviews recent advancements aimed at addressing such challenges in categories. Note that we will only focus on memory safety enforcement. Orthogonal attacks, such as side channels and transient execution attacks, are not discussed.

6.1. Fuzzing

One approach to improving the accuracy of the eBPF verifier is to test it more thoroughly. Fuzzing helps detect eBPF memory safety issues by generating random program inputs that test for vulnerabilities such as buffer overflows and invalid memory accesses. As shown in Table 2, general fuzzing tools such as Syzkaller [55] and libFuzzer [93] explore diverse paths in eBPF programs and associated kernel code. In general, fuzzer generated eBPF programs that cause issues in the kernel indicate a missing or an incorrect check in verifier. However, general fuzzers like AFL and Syzkaller need specific configurations for effective eBPF fuzzing [94]. Dedicated input generation and harnessing eBPF-kernel interactions are necessary to uncover eBPF-specific vulnerabilities.

6.1.1. Dedicated eBPF fuzzing. Many eBPF-specific fuzzers have been proposed recently, as shown in Table 2. Buzzer [95] is designed to test the eBPF verifier's logic by generating large volumes of eBPF programs, which it then passes through the verifier and executes in a running kernel to detect unsafe behavior. Buzzer's dedicated eBPF program generation strategies and instrumentation make it effective at identifying complex bugs in eBPF's safety checks. Similarly, BRF (BPF Runtime Fuzzer) [96] enhances the verification and testing of the eBPF runtime environment. Unlike conventional fuzzers, BRF is specifically customized to generate semantically correct eBPF programs that can pass the verifier's safety checks. To accomplish this, BRF incorporates an iterative, error-driven approach to enforce eBPF-specific semantics and to satisfy syscall dependencies for correct program loading, attaching, and execution. This methodology enables BRF to explore deeper execution paths within the eBPF runtime and has demonstrated its effectiveness, achieving significantly higher code coverage and detecting previously unknown vulnerabilities.

Unlike traditional fuzzers that rely primarily on crash detection, BpfChecker [97] is designed to identify issues across eBPF runtimes. BpfChecker adopts a different approach by performing differential testing, where it compares the execution states of multiple eBPF runtimes (e.g., Solana rBPF, vanilla rBPF, and Windows eBPF) to detect inconsistencies that may signal potential flaws. BpfChecker uses a specialized eBPF-specific intermediate representation (IR) and constrained mutations to produce semantically correct programs, enabling deep exploration of runtime behaviors. Additionally, LKL-based Fuzzer [53] outperforms existing solutions by leveraging the Linux Kernel Library (LKL) to run multiple lightweight kernel instances simultaneously, effectively enhancing fuzzing speed and efficiency.

6.1.2. Limitations of Fuzzing. Specialized eBPF fuzzing faces two challenges: (1) achieving a high success rate in generating verifier-approved eBPF programs and (2) effectively producing programs that trigger exploitable memory vulnerabilities. While state-of-the-art approaches such as BPFChecker struggle with verifier acceptance (i.e., 18%), BRF significantly improves on this with a 97% success rate. However, BRF's high acceptance rate exposes another crucial limitation. It reported only 1 memory error among six identified vulnerabilities, fewer than other methods. This reveals that even sophisticated semantic aware schemes such as BRF grapple with generating unsafe operations that both pass verifier checks and uncover critical memory issues.

TA 7. Fuzzing is useful in identifying vulnerabilities in the eBPF verifier. However, fuzzing is inherently incomplete due to the near-infinite state space, and eBPF presents challenges in improving the coverage and generating programs that pass verification.

6.2. Isolation

Isolation techniques have matured to efficiently separate the kernel from its submodules [98], [99], [100], leveraging program isolation analysis for automation [101], [102]. In the context of eBPF, these methods aim to restrict all memory operations of the eBPF subsystem within a limited protection domain, preventing unintended interactions with the host kernel. Software-based solutions, such as Software Fault Isolation (SFI) [103], enforce strict memory access policies to ensure that eBPF programs cannot compromise kernel memory. Meanwhile, hardware-assisted isolation, utilizing features like Intel MPK or ARM MTE, provides a hardware-enforced safeguard, restricting eBPF programs from accessing unauthorized memory regions.

6.2.1. Software-based Fault Isolation. SandBPF [36] and SafeBPF [37] enhance runtime memory safety in eBPF through SFI, confining eBPF programs to defined memory regions to prevent unauthorized access to kernel memory. SandBPF sandboxes eBPF programs using binary rewriting to manage both memory access and control-flow operations. This includes address masking for spatial memory safety and monitored control-flow transfers to ensure eBPF programs only access approved memory regions. Similarly, SafeBPF uses SFI to enforce spatial memory safety by masking all eBPF memory addresses, redirecting out-of-bounds accesses back into a sandbox. Together, these approaches strengthen eBPF's runtime memory protection, with modest overhead, shielding the kernel from spatial memory error exploits that static verification may overlook.

6.2.2. Hardware-assisted Isolation. HIVE [39] introduces a hardware-assisted isolated execution environment for eBPF on AArch64 to enhance memory safety. While eBPF programs traditionally operate with direct access to kernel memory, HIVE isolates eBPF programs as if they were independent kernel-mode applications. It achieves this by creating a dedicated "BPF space" for eBPF data, while keeping eBPF code in kernel memory, thereby ensuring that eBPF programs are fully compartmentalized and cannot access kernel memory directly. HIVE leverages hardware features (e.g., Pointer Authentication) to enforce strict access boundaries dynamically, providing robust memory safety without complex static analysis. This approach effectively prevents eBPF programs from accessing or modifying kernel memory. Additionally, MOAT [38] is designed to isolate eBPF programs using Intel Memory Protection Keys (MPK). This approach aims to prevent malicious eBPF programs from tampering with kernel memory by isolating them in a secure domain. MOAT introduces a two-layer isolation scheme, with the first layer using MPK to create distinct memory domains for eBPF programs, the kernel, and shared resources. The second layer involves isolating eBPF programs in their own address spaces.

6.2.3. Limitation of Isolation. Proposed isolation techniques effectively contain the memory accesses of eBPF programs, mitigating unauthorized access to kernel memory that are caused by memory errors in eBPF programs memory. However, these defenses primarily focus on preventing attacks originating solely from the eBPF program's memory accesses directly, some of them only focus on spatial safety. Advanced attacks usually exploit interactions between an eBPF program and the kernel, such as: (1) forging pointers in the corrupted eBPF program memory and then passing those into the kernel, potentially allowing unverified pointers to trigger harmful operations to kernel memory (e.g., arbitrary memory accesses) to enhance attacker's capability of exploitations [28], [29], and (2) by directly misusing eBPF helpers and other kernel APIs to corrupt kernel memory in unsafe ways (e.g., creating a use-after-free vulnerability, as shown in Section 4.3), also known as Cross-boundary Interface Vulnerabilities (CIVs) [40], [41], [104]. Addressing those issues will require more robust validation of memory access using cross-boundary interfaces between the kernel and eBPF program (e.g., eBPF helper functions) and deeper scrutiny of kernel-eBPF interactions, similar to the thorough security vetting needed for kernel-driver interfaces [99], [105], [106].

TA 8. Isolation confines eBPF program to reduce unauthorized kernel accesses; however, approaches rely on hardware support, incur notable overhead, and do not address risks from indirect kernel access.

6.3. Runtime Memory Safety Enforcement

Ensuring memory safety during eBPF program execution is crucial, as eBPF programs operate directly within kernel memory, where unchecked memory accesses could lead to kernel exploitation. To mitigate these risks, advanced techniques enhance eBPF verification with runtime enforcement, preventing accesses that may violate memory safety or aim to exploit memory errors.

6.3.1. Summary of Proposed Runtime Defenses. Jin et al. [107] propose a hybrid model for enforcing memory safety in eBPF by combining static analysis with selective runtime symbolic execution. Their approach begins with fast static analysis to identify potentially unsafe paths, reserving symbolic execution only for flagged paths during runtime, dynamically verifying memory access risks in real-time and reducing overhead. Alternatively, Jia et al. [108] enhance system call security within the kernel through Seccomp-eBPF, enabling eBPF to enforce policies based on current execution context. By introducing custom helper functions, their method manages complex memory interactions safely, limits exploitable code paths, and reduces the kernel's attack surface while retaining eBPF's flexibility. Interestingly, WebAssembly offers an alternative runtime memory safety model that could inform eBPF development [109]. By using a sandboxed execution approach with dynamic runtime checks, Wasm isolates untrusted code from the host environ-

ment, eliminating the need for a static verifier and enforcing strict bounds on memory and control flow.

6.3.2. Limitations of Runtime Memory Safety Enforcement. Runtime enforcement techniques offer important eBPF memory safety controls but have notable limitations. Seccomp-eBPF, while improving system call security, only partially protects memory by targeting specific system calls rather than general access, leaving gaps exploitable through other eBPF interactions. Similarly, Jin et al.'s hybrid model narrows runtime checks to risky paths but faces scalability issues with symbolic execution, potentially allowing memory errors to bypass detection. Together, these methods enhance security but do not ensure full memory safety. Approaches similar to the memory safety model in Wasm, while guaranteeing comprehensive memory safety, have significant runtime overheads that render it impractical for latency-sensitive and high-performance eBPF applications.

TA 9. Runtime checks can mitigate memory errors in eBPF programs to protect kernel, but their effectiveness is limited by incomplete coverage and the resource constraints of eBPF (discussed in Section 7.5).

6.4. Enhancing Static Validation

Ensuring the memory safety of eBPF programs requires enhanced static validation than the current eBPF verifier. Researchers have proposed a variety of enhancements to eBPF verification to improve accuracy, as shown in Table 3. However, these techniques do not ensure that all memory safety vulnerabilities can be detected (i.e., are not sound), leaving potential vulnerabilities unindentified.

6.4.1. Validating a Single eBPF Program. In the scope of validating single eBPF programs, Vishwanathan et al. [42] improve validation by enhancing tnum arithmetic precision[4], optimizing operations like addition and subtraction, and introducing an efficient multiplication algorithm, which boosts verifier performance. Follow-up works [43], [44] target formally verifying range analysis in the eBPF verifier, automating constraint generation from kernel C code and using SMT solvers for direct validation. This work also introduces *differential synthesis* to auto-generate eBPF test programs that detect semantic mismatches, revealing previously unknown bugs and establishing validation soundness. Nelson et al. [45] propose a *proof-carrying* approach that shifts verification to user-space, allowing the kernel to focus on proof-checking, thus simplifying verification.

Recently, Sun et al. [46] introduce state embedding as a method to enhance static verification in the eBPF verifier by uncovering logic bugs that may allow unsafe programs to bypass checks. Their technique embeds specific concrete states into eBPF programs, enabling the verifier to validate these states against its approximations; any mismatch signals a logic flaw. This approach enables precise bug detection without additional specifications, thus greatly improving the accuracy of eBPF verification. Similarly, HyperBee [47] proposes the notion of a *verifer chain*, which includes a set of verifiers on top of the existing eBPF verifier for additional customized verification. The verifier chain allows HyperBee to customize the security checks based on the different guest environments' specific needs and threat models, such as user-defined security policies, signature validation, and malware detection. Additionally, recent efforts such as PREVAIL [111] propose enhancing the eBPF verifier by integrating advanced static analysis techniques to reduce false classifications of memory operations, while Kflex [112] proposes replacing current eBPF extension mechanisms to enhance isolation and security policy enforcement.

6.4.2. Validating the Composition of eBPF Programs. The challenge is complicated when dealing with the composition of eBPF programs. The advantages of such composition are exemplified by BMC [48], where eBPF is used to intercept network-level requests before they reach the full network stack, resulting in substantial performance gains. However, such composition must be carefully managed to avoid compromising system safety. For example, Somaraju [110] highlights a critical issue in such compositions, noting that combining multiple eBPF programs, where all of them passed the verification individually, can still lead to stack overflows. The eBPF verifier, which checks each program in isolation, does not account for the collective influence of multiple programs interacting at runtime. One approach to prevent vulnerabilities in eBPF program composition is taken in KFuse [49], a framework that enhances both the security and performance of eBPF compositions by merging chains of eBPF programs post-verification. KFuse ensures that performance bottlenecks (e.g., indirect jumps) are mitigated without undermining the verifier's security guarantees. This highlights the dual nature of eBPF composition: it can offer significant performance improvements but requires careful management to ensure memory safety.

6.4.3. Limitations of Static Verification Techniques. Despite progress in eBPF verification, unresolved challenges persist. While static verification techniques such as verifier chains and proof-carrying code avoid runtime overhead, they add complexity and compatibility issues within the kernel. Tools such as tnum arithmetic, effective for validating single programs, fail to scale across multiple interacting eBPF programs, as essential constraints may be lost across program boundaries. Composition-oriented solutions such as KFuse, which merges verified programs to reduce indirect jump costs, assume no further memory safety issue is introduced by interaction of eBPF programs, overlooking exploits through indirect memory access across eBPF programs that are not detectable in isolated checks. BMC, meanwhile, bypasses built-in deeper validation layers within the kernel, weaken the kernel's original ability

4. In the eBPF verifier, "tnum" refers to a way of tracking uncertain or partially known values by representing each value as a combination of known and unknown bits. This representation is particularly useful for analyzing indirect memory accesses or data derived from untrusted inputs.

Approach	Key Contributions	Single Program	Composition
Enhanced Arithmetic [42]	Improves arithmetic precision and efficiency	✓	
Range Analysis [43]	Automates condition generation with SMT solvers	✓	
Proof-Carrying Verification [45]	Shifts verification to user-space	✓	
State Embedding [46]	Embeds states to detect verifier logic bugs	✓	
Verifier Chain [47]	Layers multiple verifiers for flexible validation	✓	
Spatial Safety in Composition [110]	Identifies stack overflow risks in multi-eBPF interactions		✓
Program Fusion [49]	Merges eBPF chains post-verification for security		✓

TABLE 3: Summary of eBPF memory safety approaches with focus on enhancing verification.

of validation. Thus, validating eBPF programs separately for composed eBPF programs leave unaddressed security gaps in the interdependent environments. Furthermore, all abovementioned methods, similar to the eBPF verifier, lack formal verification, leaving the potential for bypasses.

TA 10. Advanced static verifications improve eBPF memory safety. However, challenges remain as existing techniques are not sound and validate composite eBPF programs based on incomplete assumptions.

7. Summary of Root Causes

In this section, we summarize the root causes of the memory safety concerns introduced by eBPF. The aim is to explain why these issues persist, despite all existing efforts described in the previous sections, even if they were implemented perfectly.

7.1. The Dilemma of Privilege Restrictions

Requiring privileges for eBPF execution introduces a challenge of choosing between enhancing security and maintaining deployment flexibility. As discussed in Section 5, while privilege restrictions can prevent unprivileged users from exploiting the kernel through attaching malicious eBPF programs, these controls are optional and sometimes disabled for performance, allowing potential security gaps. Although limiting privileges can enhance security, it not only limits eBPF's usability in flexible deployment environments where unprivileged execution is favored for simplicity, but also cannot fully address memory safety challenges, as attackers [91], [92] may still pursue privilege escalation with limited privileges.

7.2. eBPF Verifier's Validation is Untenable

The eBPF verifier is claimed to ensure the safety, security, and correctness of eBPF programs before they are executed. However, as discussed in Section 4, memory safety checks implemented by the eBPF verifier are incomplete and unsound. The eBPF verifier's validation is widely considered untenable [31], [36], [37] due to a combination of structural and operational shortcomings that contribute to persistent vulnerabilities. One of the primary issues is its incomplete memory safety model. While the verifier provides some safeguards against spatial errors, such as bounds checking, it lacks robust mechanisms for enforcing temporal safety. This partial approach to memory safety leaves critical gaps, particularly as more complex memory management patterns emerge in eBPF programs, creating exploitable vulnerabilities that are difficult to detect or prevent at runtime.

A major factor contributing to the verifier's limitations is its reactive and unstructured evolution, which has led to a complex codebase [31] (22k lines in one file). Rather than following a proactive design strategy, the verifier has grown in an ad-hoc manner, with new features and bug fixes added piecemeal when problems arose. This uncoordinated expansion has resulted in a fragmented structure, prone to inconsistencies and increasingly challenging to validate for soundness. *The lack of formal specification leads to an unsound and incomplete verification process.* Over the years, this reactive growth has caused the verifier's codebase to expand tenfold in size, introducing complexity that makes it difficult to validate reliably. As the codebase becomes larger and more intricate, it not only obscures subtle errors but also creates numerous entry points that attackers can exploit. This escalating complexity underscores the verifier's vulnerability and highlights the challenges in ensuring robust security within an ever-growing eBPF ecosystem.

Adding to these challenges, modern compiler optimizations often generate code that conflicts with the verifier's underlying assumptions [37]. For example, optimizations from LLVM, commonly used for compiling eBPF programs, may produce instruction sequences that the verifier cannot accurately assess [113]. To bypass these conflicts, developers are often forced to make adjustments to their code specifically to meet the verifier's requirements.

7.3. eBPF Helpers Ignore Memory Safety

eBPF helpers lack memory safety checks and unconditionally trust the eBPF verifier (and vice versa). As shown in the Listings 5 in Section 4.3, the eBPF helper functions' source code contain zero security checks. eBPF helper functions are designed with the assumption that all verified eBPF programs are benign and use eBPF helpers correctly, thus, there are no security checks inside the helper functions' code [31], [37], [114]. Since the eBPF verifier is responsible for ensuring that all eBPF programs adhere to strict safety and security constraints before executing, the eBPF helper functions do not include additional memory safety checks. This design choice optimizes performance by avoiding redundant checks, relying on the initial validation to ensure that any inputs to the helper functions are already

safe. However, as described above, once a malicious eBPF program tricks the verifier to get loaded, the eBPF helper will not able to help mitigate any attempted exploits.

7.4. Isolation Defenses Are Insufficient Alone

Isolation techniques have emerged as a primary focus for addressing memory safety in eBPF, aiming to contain potentially unsafe operations within protection domain boundaries to prevent unauthorized access to kernel memory. These proposed approaches, which include both software-based and hardware-assisted solutions, focus on creating strict memory access restrictions for eBPF programs.

However, isolation alone does not ensure comprehensive memory safety for the kernel. Isolation techniques primarily address spatial memory issues [37], preventing unauthorized accesses to kernel memory through corrupted eBPF programs. Yet they often fall short in protecting against temporal memory errors, which require dynamic, real-time memory validation. Additionally, hardware-assisted isolation methods are not universally compatible or feasible, depending on what specialized hardware support is available in the deployment environments. More importantly, hardware features themselves may suffer from weaknesses that allow attacker to circumvent their protections [115], [116]. Also, as mentioned in Section 6.2.3, current isolation approaches do not account for indirect access to kernel memory. Moreover, isolation by itself does not address the security of interactions between the eBPF program and the external modules it interacts with. Any data shared with the eBPF program or any accessible functionalities must be protected according to their location, target, and the desired semantics, precisely identifying such data is required.

Thus, while isolation enhances memory safety for eBPF, it is insufficient on its own. Isolation must be supplemented by precise static checks or additional runtime memory checks to fully address memory safety comprehensively.

7.5. Low Budget for Runtime Checks

The budget for applying runtime checks to enforce memory safety in eBPF is limited, primarily due to two constraints: (1) the need for high efficiency and (2) the limits on memory usage and instruction counts. While runtime checks are a common solution for enforcing memory safety, intensive usage of runtime checks, such as described in Section 6.3.1 are not well-suited to these constraints.

First, eBPF is designed for high-performance use cases, such as networking and security monitoring, where efficiency is main goal. Introducing runtime checks would add extra processing overhead, slowing eBPF program execution. This overhead can severely impact performance, particularly in latency-sensitive tasks like packet filtering, where eBPF's strength lies in handling large data volumes with minimal delay. Even minor increases in execution time due to runtime enforcement could accumulate into significant performance bottlenecks, undermining eBPF's primary advantage in time-critical operations.

Second, eBPF programs operate under strict constraints on both memory usage [117] and instruction counts [118]. Currently, while eBPF allows a maximum of 1 million instructions to be executed at runtime, a program must be no more than 4,096 instructions long [114]. Implementing runtime checks would require instrumenting additional instructions to verify memory safety, possibly consuming the limited instruction budget and leaving less room for essential program logic. Developers are then forced to implement the intended functionality as multiple eBPF programs, making verification harder, as stated in Section 6.4. Additionally, runtime checks may add memory overhead that contradicts eBPF's requirements for minimal memory usage (e.g., 512 bytes for maximum stack size [114], [119]), making runtime checks impractical for scenarios where efficiency and resource economy are critical.

TA 11. eBPF's memory safety issues arise from inherent design trade-offs (e.g., balancing privilege restrictions, flexibillity, usability, and performance) and from incomplete safety enforcements (e.g., unsound verifier and unprotected helper functions). Additionally, the limited capacity for runtime checks and isolation further compounds the issues, leaving eBPF still susceptible to memory errors despite various countermeasures operating on multiple level.

8. Towards Memory-Safe eBPF

In this section, we aim to explore the efficacy of addressing the limitations found in current defenses and research efforts discussed in prior sections to enforce memory safety for the eBPF subsystem.

8.1. Full Memory Safety: How Far are We?

Given the expectations that eBPF programs are not supposed to tamper with kernel memory and that only unsafe memory operations may result in memory errors [120], we aim to determine how far the eBPF subsystem is from enforcing full memory safety as a guide to discuss future directions for improving eBPF security.

To estimate how far eBPF programs are from full memory safety, we apply static memory safety validation [121] to eBPF programs to quantify their unsafe operations. We can then compare memory safety threats in eBPF to normal C programs computed elsewhere [122], [123]. We aim to cover memory safety comprehensively, which includes in following three classes, defined by Huang et al. [123].

- ***Spatial Safety***: All accesses of a memory object must only access memory within the object's allocated region.
- ***Type Safety***: All accesses of a memory object must only access the same data types for each offset and each field.
- ***Temporal Safety***: All accesses of a memory object must not access the object's allocated region before allocation nor after the object's deallocation.

We compute the *fraction* of unsafe memory operations in C and eBPF programs. Specifically, we compare three categories of programs listed below.

- ***Public eBPF programs***: eBPF programs that are publicly available for various purposes. For this evaluation, we pick the eBPF programs in Linux Kernel and BCC for their broad usage and significance. The Linux kernel includes foundational eBPF programs, while BCC, the most starred (21K) eBPF-related repository on GitHub, serves as a key reference for building custom eBPF programs.
- ***Malicious eBPF programs***: eBPF programs with verified exploits to cause memory corruptions. The malicious eBPF programs in our evaluation are all extracted from verified Syzbot bug reports (serving as reproducers) or PoC of CVEs. The validation can naturally be extended to other intentionally crafted malicious eBPF programs, such as those generated by the EPF approach [29].
- ***C programs***: User-level programs that are originally evaluated by DataGuard and Uriah [122], [123].

To conduct memory safety validation, we adapt the analysis in DataGuard [122] for validating stack memory accesses comprehensively, and Uriah [123] for validating heap memory accesses for spatial and type safety. As no general static analysis for computing temporal safety violations on the heap precisely, we employ escape analysis [124] to over-approximate such cases. The over-approximation of temporally unsafe operations identified by escape analysis has only a minimal impact on results, as heap usage is uncommon within eBPF programs[5]. We manually inspected all the helpers used in the eBPF programs we tested and found no memory safety checks in any of them (e.g., Listing 5). This aligns with the claim in Section 7.3 that helpers ignore memory safety. Because of this, we did not implement our analysis in a way aware of the potential helper checks.

Our analyses are for the default Linux ecosystem. For public eBPF programs, we select all 31 programs in the Linux Kernel [127] and all 126 programs in the BCC project [128]. We selected 102 malicious eBPF programs (85 from Syzbot, 17 CVEs) for the evaluation, representing the complete set of memory safety vulnerabilities discovered in 2024 that have concrete PoCs and reproducers. Unlike the public eBPF programs selected (written in C code and aligned with LLVM compiler tool chain to be compiled into LLVM Bitcode for memory safety validation analyses), the malicious eBPF programs extracted from Syzbot reports are presented in bytecode. To make them compatible with LLVM toolchain, we used an IDA Pro plugin [129] to lift (i.e., disassemble and decompile) the eBPF bytecode to C source code. For C programs, we used all programs that are originally evaluated in DataGuard and Uriah papers; the goal is to use such data to serve as an comparison of unsafe operations between eBPF programs and C programs.

However, precisely validating memory safety for eBPF programs requires *extracting kernel-specific constraints*, such as the size of kernel data accessed by eBPF (e.g., eBPF maps). The state-of-the-art static memory safety validation approaches that we utilized [122], [123] lack support for this cross-boundary constraint extraction, so constraints from the kernel are not recognized by the memory safety validation. To assess the impact of these kernel constraints, we measured the fraction of unsafe operations under two scenarios: (1) computing the *upper bound* of unsafe operations by identifying all pointers that could possibly violate one of the three classes of memory safety in eBPF when associated kernel constraints are available and (2) computing the *lower bound* of unsafe operations by finding operations that would be unsafe under any kernel constraints (i.e., use variable lengths and perform unsafe type casts). Note that, we do not classify all pointer arithmetic or type cast operations as unsafe, for the ones whose constraints can be retrieved with eBPF program for them to be validated as safe, such as the constant size or the concrete types, we classify such operation as safe by leveraging DataGuard/Uriah analyses. In other words, the availability of the kernel constraints are the only cause of the gap between the lower and upper bound of unsafe operations. This reveals the potential to close the gap by pushing the upper bound toward the lower bound through effectively extracting kernel constraints.

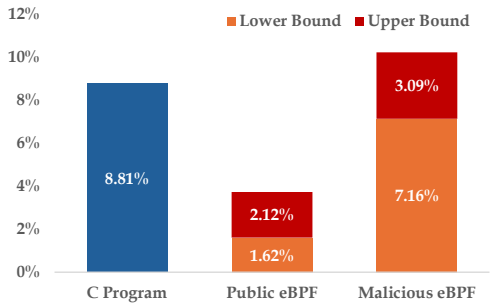

Figure 2: Fraction of Unsafe Memory Operations.

As shown in Figure 2, the fraction of unsafe memory operations in public eBPF programs is **relatively low**, given the intense nature of memory accesses of eBPF programs for their functionality. While 8.81% of memory operations may be unsafe in the tested C programs, only 1.62%-3.74% of memory operations would be unsafe in public eBPF programs. Even the fraction of unsafe operations in malicious (syzbot-generated) eBPF programs is comparable to C programs, which we attribute to the fact that (1) the eBPF verifier limits the number of unsafe operations, *even when generating malicious eBPF programs*, and (2) they are built solely for exploiting memory errors (e.g., by Syzbot), and thus do not contain many meaningful operations for functionalities. Based on these, the higher fraction of unsafe operations in malicious eBPF programs is expected.

Notably, although malicious eBPF programs have a comparable fraction of unsafe operations as C programs, the number of individual unsafe operations is orders of magnitude smaller. For example, nginx (one of the evaluated C programs that has a smaller fraction of unsafe operations than average) has more than two million instructions in total of which around 30% are memory operations, where

5. eBPF programs cannot manage heap memory internally [125], [126]. The data that they generally access are on the kernel heap, e.g., eBPF maps.

		eBPF-Only			Shared Objs		
		Spatial	**Type**	**Temp**	**Spatial**	**Type**	**Temp**
Public		33.5%	2.1%	5.8%	48.9%	2.5%	7.2%
Malicious		31.1%	5.3%	8.1%	43.7%	5.6%	6.0%
eBPF Verifier [30]	V	◐	◕	◕	◐	◕	○
HyperBee [47]	V	◐	◕	○	○	○	○
KFuse [49]	V	◐	◕	○	◐	◕	○
PREVAIL [111]	V	◐	◕	◕	◐	◕	○
SandBPF [36]	II	◕	○	○	◕	○	○
SafeBPF [37]	II	◕	○	○	◕	○	○
HIVE [39]	II	◕	○	○	◕	○	○
MOAT [38]	II	◕	○	○	◕	○	○
Prevail2Radius [107]	T	◕	◕	○	◕	◕	○
Seccomp-eBPF [130]	T	◕	○	○	◕	○	○
TnumArith [43]	T	◕	○	○	○	○	○
RangeAnalysis [44]	T	◕	○	○	○	○	○

TABLE 4: Protection Scope of Defenses, "V" stands for validation approaches, "II" stands for isolation approaches, "T" stands for target defenses for specific partterns.

58,347 of them are unsafe memory operations. For all evaluated eBPF programs, recalling that the maximum instruction count per eBPF program is 4,096 (see Section 7.5), the average number of unsafe memory operations is only 37 (lower bound) and 85 (upper bound) for public eBPF programs, and 29 (lower bound) to 41 (upper bound) for malicious eBPF programs[6]. The drastically smaller number of unsafe memory operations in eBPF programs compared to general C programs, suggests that less effort (i.e., fewer runtime checks and/or code modifications) is necessary to prevent unsafe memory operations from exploiting the eBPF subsystem than general C programs comprehensively.

8.2. Protection Scope of Existing Defenses

To better understand the limitations of existing defenses, we categorize each unsafe operation identified by the static memory safety validation (Section 8.1) along two dimensions. First, we classify the type of memory safety violation as either *spatial*, *type*, or *temporal*. Second, we determine whether the operation accesses memory local to the eBPF program (*eBPF-only*) or interacts with memory shared with the kernel (*shared*). We also distinguish between *public* and *malicious* eBPF programs. This categorization provides the basis for systematically evaluating the scope of defenses.

Table 4 presents our evaluation of these defenses. The top portion of the table reports the distribution of unsafe operations by memory safety category and whether the operations target eBPF-only objects or objects shared with the kernel. This breakdown highlights that a significant portion of unsafe operations involve shared memory, which many defenses fail to adequately protect. The bottom portion of the table lists defenses grouped by their primary strategy: **verification** for static/dynamic/symbolic validation, **isolation** for isolation approach, or **targeted** for defenses designed to mitigate specific bug patterns. Our evaluation combines experimental evidence and fundamental design analysis.

The legend used in the table are:

- "◕" indicates that the defense attempts comprehensive coverage of a violation category but relies on ad hoc checks that may be *unsound*, missing attack vectors.
- "◐" denotes defenses that confine accesses to eBPF-only memory but do *not* protect data shared with the kernel, leaving part of the attack surface unguarded.
- "◴" represents targeted defenses that address specific vulnerabilities, without providing general coverage for the entire class of violations.
- "○" means no protection is provided for that category.

Clearly, none of the defenses, whether currently deployed or proposed in research, fully or soundly cover any category of unsafe operations. Therefore, we do not include a legend symbol representing complete protection.

8.3. Enhancing Static Memory Safety Validation

The limitations of current defenses (discussed in Section 6 and Section 8.2) highlight a need for a stronger focus on improving the eBPF verifier's capability to validate memory safety before the eBPF program is loaded into kernel. Approaches such as DataGuard and Uriah [122], [123] and the Rust compiler provide valuable insights for enhancing memory safety validation, enabling the prevention of a wide range of memory error exploitations without relying on costly runtime checks or incomplete isolation techniques.

The hypothesis of static memory safety validation is that most objects in a program are only accessed via safe memory operations across all aliases, enabling comprehensive and efficient memory safety enforcement through isolation, as examined in DataGuard and Uriah [122], [123]. In Figure 2, the gap between the upper and lower bounds of unsafe operations is significant, with the upper bound exceeding the lower bound by more than a factor of two for public eBPF programs and showing over a 40% increase for malicious programs. This discrepancy highlights the conservativeness of static analysis in the absence of complete kernel information. To address this, we performed a conservative targeted analysis that extracts kernel constraints from only the associated submodule using available information exist in the eBPF program ((e.g., header files, map definitions)). This refined analysis reduced the proportion of unsafe operations to 1.74% for public and 8.63% for malicious eBPF programs for upper bound, a 95% and 53% drop respectively from the gap between original lower and upper bounds, demonstrating the importance of incorporating kernel constraints to improve the precision of static memory safety validation.

A promising approach lies in extending the memory safety validation in the eBPF verifier to incorporate the kinds of memory safety guarantees offered by languages like Rust [131]. Rust's memory model, which ensures strict ownership [132], borrowing [133], and lifetimes [134], could serve as a blueprint for enhancing the eBPF verifier's static analysis capabilities. However, the challenge of extracting kernel-specific constraints lies in accurately identifying

6. Malicious eBPF programs are smaller than public ones, as they focus solely on exploiting vulnerabilities without actual functionalities, resulting in fewer unsafe operations but a higher proportion overall.

aliases and the memory objects that are accessed across eBPF and kernel boundaries, as mentioned in Section 8.1. Alias analysis, which determines whether two pointers reference the same memory location, becomes complex in the context of eBPF because pointers may interact with both kernel and user data through helper functions. These interactions complicate the verifier's ability to maintain precise safety constraints, as it must account for indirect accesses caused by helper functions or shared kernel data structures. To address this, a refined alias-tracking mechanism (such as combining data-flow-based and type-based alias analysis [135], [136], [137]), and an analysis that can determine the data that is shared between eBPF program and kernel (e.g., KSplit [99]) need to be integrated into the validation.

Moving forward, syntactic annotations (e.g., Checked-C [138], [139] and Rust extensions) could be employed to improve the accuracy of memory safety validation. Although providing complete memory safety guarantees for all eBPF programs is challenging, the verifier could adopt a fallback approach: rejecting any programs that lack sufficient safety information, thereby prompting developers to refine their code to meet validation requirements. This fallback model, successfully implemented in frameworks such as CRT-C [140] and EC [141] for privilege separation and program compartmentalization, supports safer development by combining static analysis with developer responsibility. Applying the same model to memory safety validation would not only enhance security but also create a development process that encourages careful memory handling in eBPF, leading to a more secure codebase. As mentioned in Section 8.1, the adopted analyses are unaware of helper checks, which may lead to overapproximated results by flagging unsafe operations that are actually prevented by existing checks. Enhancing the analysis to recognize such checks and validations is a promising direction for future work, as demonstrated by [142].

Meanwhile, recent efforts such as PREVAIL [111] offer promising techniques for augmenting or even replacing the default eBPF verifier with a more robust validation mechanism. PREVAIL's modular design allows for the incorporation of language-based safety properties similar to those provided by Rust, further strengthening eBPF security. Moreover, we consider the approach of Kflex [112], which proposes an alternative extension mechanism to the current eBPF framework. By substituting existing extension mechanisms with Kflex, we can facilitate stronger isolation and more precise enforcement of security policies at the boundary between eBPF programs and kernel code.

8.4. Protecting eBPF from Unsafe Operations

Enhancing memory safety validation offers a robust solution for identifying unsafe memory operations that need attention. Additional memory safety enforcement can be employed specifically to unsafe operations.

8.4.1. Adding Runtime Checks Strategically. Deploying intensive runtime checks on every memory operation is infeasible given the nature of eBPF (see Section 7.5). While deploying runtime checks extensively is impractical for eBPF due to performance constraints, strategically placed runtime checks can still be beneficial in high-risk scenarios. By selectively applying these checks to the operations that are most vulnerable, eBPF can achieve a balance between safety and efficiency. As discussed in Section 8.1, the low number and fraction of unsafe memory operations in public eBPF programs suggests that only a modest number of runtime checks may be necessary, which may limit the impact of these overheads while enforcing memory safety. Use of the Uriah system [123] for heap memory safety validation was shown to reduce the overhead of runtime checks commensurate with the fraction of runtime checks removed from the objects found safe.

Complementing this, runtime checks can be restricted to critical interactions with sensitive kernel data, ensuring essential protections are in place without compromising performance. This layered approach, combining precise static validation with selectively placed runtime checks, could secure kernel memory safety with efficient use of computational resources. Note that, though fuzzing is generally considered incomplete in terms of coverage, effective fuzzing can be introduced using a similar insight that only the unsafe operations need to be targeted for testing. We can explore how to generate inputs from potentially malicious eBPF programs likely to trigger errors. Such directed fuzzing approaches can more precisely generate eBPF inputs that both pass the eBPF verifier and trigger vulnerabilities, enhancing the detection of memory vulnerabilities.

8.4.2. Advancing Finer-grained Isolation. Current isolation techniques primarily focus on preventing kernel memory corruption directly from eBPF memory, but often overlook other threats, such as the malicious use of unchecked eBPF helpers (discussed in Section 6.2.3). These cross-boundary interface vulnerabilities (CIVs) have been demonstrated in recent studies, notably in CIVScope [40] and ConfFuzz [104], where attackers forge pointers or misuse helper functions to indirectly compromise kernel memory.

A finer-grained isolation model is needed to address these vulnerabilities more precisely. Memory safety validation can play a crucial role in guiding the application of isolation measures. For example, eBPF helpers that perform inherently safe memory operations need not be isolated, whereas operations flagged as unsafe could be sandboxed or subjected to additional runtime checks. This targeted approach not only tightens security where it is most needed but also avoids the performance penalties associated with blanket isolation. Moreover, emerging hardware-assisted isolation technologies [38], [39] offer promising avenues to reinforce these defenses. Unexplored hardware features such as ARM's Memory Tagging Extension (MTE) [143] and the CHERI (Capability Hardware Enhanced RISC Instructions) [144] architecture provide mechanisms for enforcing strict memory access boundaries, facilitating more dynamic, fine-grained isolation schemes.

8.4.3. Migrating to Memory Safe Language. Migrating eBPF programs to a memory-safe language like Rust, rather than relying solely on the verifier to catch errors, can be an effective way to fundamentally enhance memory safety in the eBPF ecosystem. Jia et al. [31] highlight that while the eBPF verifier is essential for safety, it cannot fully address memory safety challenges, especially when multiple eBPF programs or complex interactions are involved. Shifting to Rust-based eBPF programs could sidestep many of these issues by embedding runtime memory safety checks directly into the programs at compile time, reducing the reliance on separate runtime verification. With Rust's strict memory and ownership rules, vulnerabilities common in C could be minimized or eliminated, allowing eBPF to maintain high performance while ensuring stronger safety guarantees.

However, integrating Rust for eBPF introduces challenges. Supporting Rust in the Linux kernel requires adapting dependencies of Rust tooling and ensuring compatibility without adding runtime overhead (e.g., integrating a Rust compiler to the kernel or its trusted computing base). For example, Aya [145] enables writing eBPF programs in Rust but still relies entirely on the standard eBPF compilation and verification pipeline. Additionally, Rust's safety guarantees introduce extra instructions that might push eBPF programs to their instruction limits. Despite these obstacles, Rust could offer a substantial security improvement for eBPF, but successful integration requires careful handling of the above-mentioned issues.

9. Conclusion

eBPF has brought transformative capabilities to the Linux kernel, but it also introduces critical memory safety threats. We conducted the first comprehensive analysis of the root causes of eBPF's memory safety issues, revealing limitations in existing defenses and bug detection approaches, such as the verifier, isolation mechanisms, fuzzing, and runtime checks. Our findings indicate that current solutions are inadequate in ensuring comprehensive memory safety. We propose future directions for addressing these gaps, focusing on improving static validation, refining isolation, learning and borrowing from memory-safe languages, and strategically deploying runtime checks. These directions aim to guide future developments toward achieving comprehensive memory safety without compromising the performance and flexible functionality requirements for eBPF.

Acknowledgment

We would like to thank our Shepherd and anonymous reviewers for their constructive suggestions and insightful feedback. Special thanks to Tao Lyu, Aditya Basu, and Yongzhe Huang for their invaluable input on the eBPF context. This work was supported by NSF CNS-1801534, the European Research Council (ERC) Horizon 2020 grant 850868 (Funded by the European Union), and SNSF PCEGP2_186974. Any opinions, findings, conclusions, or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the U.S. Government, the NSF, the European Union, the European Research Council Executive Agency, or the SNSF. These views should not be interpreted as official policies or endorsements, expressed or implied, of the above government, organizations, or their affiliates.

References

[1] eBPF.io, *eBPF*, https://ebpf.io/.

[2] B. Gregg, "BPF: A New Type of Software," 2019, accessed: 2023-11-01. [Online]. Available: https://www.brendangregg.com/blog/2019-12-02/bpf-a-new-type-of-software.html

[3] S. Miano, M. Bertrone, F. Risso, M. Tumolo, and M. V. Bernal, "Creating complex network services with ebpf: Experience and lessons learned," in *2018 IEEE 19th International Conference on High Performance Switching and Routing (HPSR)*, 2018, pp. 1–8.

[4] M. Xhonneux, F. Duchene, and O. Bonaventure, "Leveraging ebpf for programmable network functions with ipv6 segment routing," in *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies (CoNEXT '18)*, 2018.

[5] M. A. M. Vieira, M. S. Castanho, R. D. G. Pacífico, E. R. S. Santos, E. P. M. C. Júnior, and L. F. M. Vieira, "Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications," *ACM Comput. Surv.*, vol. 53, no. 1, Feb. 2020.

[6] C. Liu, Z. Cai, B. Wang, Z. Tang, and J. Liu, "A protocol-independent container network observability analysis system based on ebpf," in *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*, 2020, pp. 697–702.

[7] K. Suo, Y. Zhao, W. Chen, and J. Rao, "vnettracer: Efficient and programmable packet tracing in virtualized networks," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, 2018, pp. 165–175.

[8] B. Gregg, *BPF Performance Tools: Using eBPF for System Performance Monitoring*, 2020.

[9] Elastic Observability Labs, "Elastic Universal Profiling Agent Now Open Source," 2023, accessed: 2023-11-01. [Online]. Available: https://www.elastic.co/observability-labs/blog/elastic-universal-profiling-agent-open-source

[10] C. Cassagnes, L. Trestioreanu, C. Joly, and R. State, "The rise of ebpf for non-intrusive performance monitoring," in *2020 IEEE/IFIP Network Operations and Management Symposium*, 2020.

[11] D. C. Panaite, "Evaluating the performance of ebpf-based security software in a virtualized 5g cluster," 2024. [Online]. Available: https://webthesis.biblio.polito.it/31760/

[12] S. Sundberg, "Evolved passive ping: Passively monitor network latency from within the kernel," 2024. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1852360/FULLTEXT02.pdf

[13] G. Fournier, "Return to sender: Detecting kernel exploits with ebpf," in *Blackhat USA*, 2022. [Online]. Available: https://i.blackhat.com/USA-22/Wednesday/US-22-Fournier-Return-To-Sender.pdf

[14] HardenedVault, "Ved-ebpf: Kernel exploit and rootkit detection using ebpf," 2023. [Online]. Available: https://github.com/hardenedvault/ved-ebpf

[15] Z. Wang, Y. Guang, Y. Chen, Z. Lin, M. Le, D. K. Le, D. Williams, X. Xing, Z. Gu, and H. Jamjoom, "SeaK: Rethinking the design of a secure allocator for OS kernel," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[16] Z. Wang, Y. Chen, and Q. Zeng, "PET: Prevent discovered errors from being triggered in the linux kernel," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[17] H. J. Hadi, M. Adnan, Y. Cao, F. B. Hussain, N. Ahmad, M. A. Alshara, and Y. Javed, "ikern: Advanced intrusion detection and prevention at the kernel level using ebpf," *Technologies*, 2024.

[18] M. Craun, K. Hussain, U. Gautam, Z. Ji, T. Rao, and D. Williams, "Eliminating ebpf tracing overhead on untraced processes," in *Proceedings of the ACM SIGCOMM 2024 Workshop on EBPF and Kernel Extensions (eBPF '24)*, 2024.

[19] J. Jia, M. V. Le, S. Ahmed, D. Williams, H. Jamjoom, and T. Xu, "Fast (trapless) kernel probes everywhere," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, 2024.

[20] L. Caviglione, W. Mazurczyk, M. Repetto, A. Schaffhauser, and M. Zuppelli, "Kernel-level tracing for detecting stegomalware and covert channels in linux environments," *Computer Networks*, 2021.

[21] B. Sharma and D. Nadig, "ebpf-enhanced complete observability solution for cloud-native microservices," in *IEEE International Conference on Communications*, 2024.

[22] G. Budigiri, "Secure and scalable policy management in cloud native networking," in *Proceedings of the 24th International Middleware Conference (Middleware '23)*, 2023.

[23] Y. He, R. Guo, Y. Xing, X. Che, K. Sun, Z. Liu, K. Xu, and Q. Li, "Cross container attacks: The bewildered eBPF on clouds," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023.

[24] D. Jin, A. J. Gaidis, and V. P. Kemerlis, "BeeBox: Hardening BPF against transient execution attacks," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[25] O. Kirzner and A. Morrison, "An analysis of speculative type confusion vulnerabilities in the wild," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021.

[26] H. Sun, Y. Xu, J. Liu, Y. Shen, and N. Guan, "Finding correctness bugs in ebpf verifier with structured and sanitized program," in *Proceedings of the European Conference on Computer Systems (EuroSys)*, 2024.

[27] M. Liber, "The good, bad and compromisable aspects of linux ebpf," 2022, accessed: 2024-10-22. [Online]. Available: https://pentera.io/wp-content/uploads/2022/07/pentera-labs-the-good-bad-and-compromisable-aspects-of-linux-ebpf.pdf

[28] Q. Liu, W. Shen, J. Zhou, Z. Zhang, J. Hu, S. Ni, K. Lu, and R. Chang, "Interp-flow hijacking: Launching non-control data attack via hijacking ebpf interpretation flow," in *29th European Symposium on Research in Computer Security*, 2024.

[29] D. Jin, V. Atlidakis, and V. P. Kemerlis, "EPF: Evil packet filter," in *2023 USENIX Annual Technical Conference (ATC' 23)*, 2023.

[30] eBPF Verifier, 2024, https://docs.kernel.org/bpf/verifier.html.

[31] J. Jia, R. Sahu, A. Oswald, D. Williams, M. V. Le, and T. Xu, "Kernel extension verification is untenable," in *Proceedings of the 19th Workshop on Hot Topics in Operating Systems (HotOS)*, 2023.

[32] Syzkaller, "Syzkaller upstream dashboard." [Online]. Available: https://syzkaller.appspot.com/upstream

[33] Y. Song, "bpf: Fix a sdiv overflow issue," 2024. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=7dd34d7b7dcf9309fc6224caf4dd5b35bedddcb7

[34] A. Nakryiko, "bpf: add support for open-coded iterator loops," 2024, accessed: 2024-10-22. [Online]. Available: https://git.kernel.org/pub/scm/linux/kernel/git/stable/linux.git/commit/?id=06accc8779c1d558a5b5a21f2ac82b0c95827ddd

[35] S. Team, "Bpf subsystem overview and open issues," 2024, accessed: 2024-10-22. [Online]. Available: https://syzkaller.appspot.com/upstream/s/bpf#open

[36] S. Y. Lim, X. Han, and T. Pasquier, "Unleashing Unprivileged eBPF Potential with Dynamic Sandboxing," in *SIGCOMM Workshop on eBPF and Kernel Extensions*. ACM, 2023.

[37] S. Y. Lim, T. Prasad, X. Han, and T. Pasquier, "SafeBPF: Hardware-assisted Defense-in-depth for eBPF Kernel Extensions," in *Cloud Computing Security Workshop (CCSW)*, Sep. 2024, in Cloud Computing Security Workshop (CCSW).

[38] H. Lu, S. Wang, Y. Wu, W. He, and F. Zhang, "MOAT: Towards safe BPF kernel extension," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[39] P. Zhang, C. Wu, X. Meng, Y. Zhang, M. Peng, S. Zhang, B. Hu, M. Xie, Y. Lai, Y. Kang, and Z. Wang, "HIVE: A hardware-assisted isolated execution environment for eBPF on AArch64," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[40] Y. Chien, V.-A. Bădoiu, Y. Yang, Y. Huo, K. Kaoudis, H. Lefeuvre, P. Olivier, and N. Dautenhahn, "Civscope: Analyzing potential memory corruption bugs in compartment interfaces," in *Proceedings of the 1st Workshop on Kernel Isolation, Safety and Verification*, 2023.

[41] Y. Huang, K. Huang, M. Ennis, V. Narayanan, A. Burtsev, T. Jaeger, and G. Tan, "Sok: Understanding the attack surface in device driver isolation frameworks," 2024.

[42] H. Vishwanathan, M. Shachnai, S. Narayana, and S. Nagarakatte, "Sound, precise, and fast abstract interpretation with tristate numbers," in *Proceedings of the 20th IEEE/ACM International Symposium on Code Generation and Optimization (CGO '22)*, 2022.

[43] S. Bhat and H. Shacham, "Formal verification of the linux kernel ebpf verifier range analysis," 2022. [Online]. Available: https://sanjit-bhat.github.io/assets/pdf/ebpf-verifier-range-analysis22.pdf

[44] H. Vishwanathan, M. Shachnai, S. Narayana, and S. Nagarakatte, "Verifying the verifier: ebpf range analysis verification," in *Computer Aided Verification (CAV): 35th International Conference*, 2023.

[45] L. Nelson, "A proof-carrying approach to building correct and flexible in-kernel verifiers," 2021, presented at the Linux Plumbers Conference 2021.

[46] H. Sun and Z. Su, "Validating the eBPF verifier via state embedding," in *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, 2024.

[47] Y. Wang, D. Li, and L. Chen, "Seeing the invisible: Auditing ebpf programs in hypervisor with hyperbee," in *Proceedings of the 1st Workshop on EBPF and Kernel Extensions (eBPF '23)*, 2023.

[48] Y. Ghigoff, J. Sopena, K. Lazri, A. Blin, and G. Muller, "BMC: Accelerating memcached using safe in-kernel caching and pre-stack processing," in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, 2021.

[49] H.-C. Kuo, K.-H. Chen, Y. Lu, D. Williams, S. Mohan, and T. Xu, "Verified programs can party: optimizing kernel extensions via post-verification merging," in *Proceedings of the Seventeenth European Conference on Computer Systems (EuroSys '22)*, 2022.

[50] eBPF Documentation, 2024, https://docs.kernel.org/bpf/.

[51] K. Recipes, "Cves are dead, long live the cve," 2019. [Online]. Available: https://kernel-recipes.org/en/2019/talks/cves-are-dead-long-live-the-cve/

[52] J. Corbet, "How kernel cve numbers are assigned," 2024. [Online]. Available: https://lwn.net/Articles/978711/

[53] M. H. N. Mohamed, X. Wang, and B. Ravindran, "Understanding the security of linux ebpf subsystem," in *Proceedings of the 14th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '23)*, 2023.

[54] S. Team, "Syzkaller subsystems overview," 2024. [Online]. Available: https://syzkaller.appspot.com/upstream/subsystems

[55] S. Contributors, "Syzkaller: Coverage-guided kernel fuzzer," 2024. [Online]. Available: https://github.com/google/syzkaller

[56] MITRE, "Cve-2024-41003," 2024, accessed: 2024-10-22. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2024-41003

[57] ——, "Cve-2024-42072," 2024, accessed: 2024-10-22. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2024-42072

[58] ——, "Cve-2024-50164," 2024, accessed: 2024-10-22. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2024-50164

[59] ——, "Cve-2024-36937," 2024, accessed: 2024-10-22. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2024-36937

[60] ——, "Cve-2024-26885," 2024, accessed: 2024-10-22. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2024-26885

[61] ——, "Cve-2024-45020," 2024, accessed: 2024-10-22. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2024-45020

[62] ——, "Cve-2024-38566," 2024, accessed: 2024-10-22. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2024-38566

[63] ——, "Cve-2024-42151," 2024, accessed: 2024-10-22. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2024-42151

[64] ——, "Cve-2024-43837," 2024, accessed: 2024-10-22. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2024-43837

[65] ——, "Cve-2024-43910," 2024, accessed: 2024-10-22. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2024-43910

[66] ——, "Cve-2024-49861," 2024, accessed: 2024-10-22. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2024-49861

[67] ——, "Cve-2024-50063," 2024, accessed: 2024-10-22. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2024-50063

[68] ——, "Cve-2024-26611," 2024, accessed: 2024-10-22. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2024-26611

[69] ——, "Cve-2024-43838," 2024, accessed: 2024-10-22. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2024-43838

[70] ——, "Cve-2024-42063," 2024, accessed: 2024-10-22. [Online]. Available: https://www.cve.org/CVERecord?id=CVE-2024-42063

[71] Linux-Kernel, "ebpf verifier: Ensuring safety and correctness in ebpf programs," 2024, accessed: 2024-10-22. [Online]. Available: https://docs.kernel.org/bpf/verifier.html#pruning

[72] NVD, "Cve-2023-2163," 2023. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2023-2163

[73] GoogleBugHunters, "A deep dive into cve-2023-2163: How we found and fixed an ebpf linux kernel vulnerability," 2023, accessed: 2024-10-22. [Online]. Available: https://bughunters.google.com/blog/6303226026131456/a-deep-dive-into-cve-2023-2163-how-we-found-and-fixed-an-ebpf\-linux-kernel-vulnerability

[74] GoogleSecurityResearch, "Proof of concept for cve-2023-2163," 2023, accessed: 2024-10-22. [Online]. Available: https://github.com/google/security-research/tree/master/pocs/linux/cve-2023-2163

[75] NVD, "Cve-2021-4204," 2021. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2021-4204

[76] Linux-Kernel, "Linux kernel ebpf verifier test suite, test_verifier.c," 2024. [Online]. Available: https://github.com/torvalds/linux/blob/master/tools/testing/selftests/bpf/test_verifier.c#L426

[77] ——, "kfuncs: Exposing kernel functions to bpf programs," 2024. [Online]. Available: https://docs.kernel.org/bpf/kfuncs.html

[78] Y. Zhang, "The secure path forward for ebpf runtime: Challenges and innovations." [Online]. Available: https://medium.com/@yunwei356/the-secure-path-forward-for-ebpf-runtime-challenges-and-innovations-968f9d71fc16

[79] Syzkaller, "KMSAN: uninit-value in dev_map_lookup_elem." [Online]. Available: https://syzkaller.appspot.com/bug?extid=1a3cf6f08d68868f9db3

[80] ——, "Reproduction program for syzkaller bug." [Online]. Available: https://syzkaller.appspot.com/x/repro.c?x=12e081f1180000

[81] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv, "Simple and precise static analysis of untrusted linux kernel extensions," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*, 2019.

[82] M. Kerrisk, 2023. [Online]. Available: https://man7.org/linux/man-pages/man7/capabilities.7.html

[83] The Linux Kernel Documentation, *LSM BPF Programs*, 2023. [Online]. Available: https://docs.kernel.org/bpf/prog_lsm.html

[84] ——, *BPF Type Format (BTF)*, 2023. [Online]. Available: https://docs.kernel.org/bpf/btf.html

[85] LWN.net, "Bpf and security," 2024. [Online]. Available: https://lwn.net/Articles/946389/

[86] SUSE, "Security hardening: Use of ebpf by unprivileged users has been disabled by default," 2023. [Online]. Available: https://www.suse.com/support/kb/doc/?id=000020545

[87] J. Corbet, "Reconsidering unprivileged bpf," 2019. [Online]. Available: https://lwn.net/Articles/796328/

[88] M. D. Verde, "Cap_bpf - a new linux kernel capability for bpf," 2023. [Online]. Available: https://mdaverde.com/posts/cap-bpf/

[89] Cilium, "Cilium: ebpf-based networking, observability, security," 2024. [Online]. Available: https://cilium.io/

[90] RedHat, "Using ebpf in unprivileged pods." [Online]. Available: https://next.redhat.com/2023/07/18/

[91] MITRE, "Cve-2023-52920: Null pointer dereference vulnerability in linux kernel's ebpf," 2024. [Online]. Available: https://nvd.nist.gov/vuln/detail/CVE-2023-52920

[92] T. Lyu, "[patch] incorrect backtracking for load/store or atomic ops," Linux Kernel Mailing List, 2023. [Online]. Available: https://lore.kernel.org/bpf/20231020155842.130257-1-tao.lyu@epfl.ch/

[93] LibFuzzer. [Online]. Available: https://llvm.org/docs/LibFuzzer.html

[94] S. Scannell, "ebpf fuzzing: Architecture, challenges, and vulnerabilities," 2020, accessed: 2024-10-22. [Online]. Available: https://scannell.io/posts/ebpf-fuzzing/

[95] Google, "A new way to fuzz for ebpf vulnerabilities," 2023. [Online]. Available: https://security.googleblog.com/2023/05/introducing-new-way-to-buzz-for-ebpf.html

[96] H.-W. Hung and A. Amiri Sani, "Brf: Fuzzing the ebpf runtime," *Proc. ACM Softw. Eng.*, vol. 1, no. FSE, Jul. 2024.

[97] C. Peng, L. Wu, M. Jiang, and Y. Zhou, "Toss a fault to bpfchecker: Revealing implementation flaws for ebpf runtimes with differential fuzzing," in *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2024.

[98] V. Narayanan, Y. Huang, G. Tan, T. Jaeger, and A. Burtsev, "Lightweight kernel isolation with virtualization and VM functions," in *Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE)*, 2020.

[99] Y. Huang, V. Narayanan, D. Detweiler, K. Huang, G. Tan, T. Jaeger, and A. Burtsev, "KSplit: Automating device driver isolation," in *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2022.

[100] X. Chen, Z. Li, T. Jain, V. Narayanan, and A. Burtsev, "Limitations and opportunities of modern hardware isolation mechanisms," in *2024 USENIX Annual Technical Conference (ATC '24)*, 2024.

[101] S. Liu, D. Zeng, Y. Huang, F. Capobianco, S. McCamant, T. Jaeger, and G. Tan, "Program-mandering: Quantitative privilege separation," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS '19')*, 2019.

[102] S. Liu, G. Tan, and T. Jaeger, "PtrSplit: Supporting general pointers in automatic program partitioning," in *Proceedings of the 24th Conference on Computer and Communications Security (CCS)*, 2017.

[103] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham, "Efficient software-based fault isolation," in *Proceedings of the Fourteenth Symposium on Operating Systems Principles (SOSP '93')*, 1993.

[104] H. Lefeuvre, V.-A. Bădoiu, Y. Chen, F. Huici, N. Dautenhahn, and P. Olivier, "Assessing the impact of interface vulnerabilities in compartmentalized software," in *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2023.

[105] V. Narayanan, T. Huang, D. Detweiler, D. Appel, Z. Li, G. Zellweger, and A. Burtsev, "Redleaf: Isolation and communication in a safe operating system," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI '20)*, 2020.

[106] A. Burtsev, V. Narayanan, Y. Huang, K. Huang, G. Tan, and T. Jaeger, "Evolving operating system kernels towards secure kernel-driver interfaces," in *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*, 2023.

[107] G. Jin, J. Li, and G. Briskin, "Research report: Enhanced ebpf verification and ebpf-based runtime safety protection," in *2024 IEEE Security and Privacy Workshops (SPW)*, 2024.

[108] J. Jia, Y. Zhu, D. Williams, A. Arcangeli, C. Canella, H. Franke, T. Feldman-Fitzthum, D. Skarlatos, D. Gruss, and T. Xu, "Programmable system call security with ebpf," 2023. [Online]. Available: https://arxiv.org/abs/2302.10366

[109] J. Dejaeghere, B. Gbadamosi, T. Pulls, and F. Rochet, "Comparing security in ebpf and webassembly," in *Proceedings of the 1st Workshop on EBPF and Kernel Extensions (eBPF '23)*, 2023.

[110] S. R. Somaraju, "In the era of linux being omnipresent, the demand for dynamically extending kernel capabilities," Master's thesis, Virginia Tech, 2023. [Online]. Available: https://vtechworks.lib.vt.edu/items/eaf0cecf-3a21-491f-a4e3-c35e5110b622

[111] E. Gershuni, N. Amit, A. Gurfinkel, N. Narodytska, J. A. Navas, N. Rinetzky, L. Ryzhyk, and M. Sagiv, "Simple and precise static analysis of untrusted linux kernel extensions," in *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '19)*, 2019.

[112] K. K. Dwivedi, R. Iyer, and S. Kashyap, "Fast, flexible, and practical kernel extensions," in *Proceedings of the ACM SIGOPS 30th Symposium on Operating Systems Principles (SOSP '24)*, 2024.

[113] J. Corbet, "The challenge of compiling for verified architectures," 2023. [Online]. Available: https://lwn.net/Articles/946254/

[114] A. Dinaburg, "Pitfalls of relying on ebpf for security monitoring (and some solutions)," 2023. [Online]. Available: https://blog.trailofbits.com/2023/09/25/pitfalls-of-relying-on-ebpf-for-security-monitoring-and-some-solutions/

[115] J. Ravichandran, W. T. Na, J. Lang, and M. Yan, "PACMAN: Attacking arm pointer authentication with speculative execution," in *Proceedings of the 49th Annual International Symposium on Computer Architecture (ISCA '22)*, 2022.

[116] J. Kim, J. Park, S. Roh, J. Chung, Y. Lee, T. Kim, and B. Lee, "TIKTAG: Breaking arm's memory tagging extension with speculative execution," in *IEEE Symposium on Security and Privacy*, 2025.

[117] I. Contributors, "Resource limits in ebpf on linux," 2024. [Online]. Available: https://docs.ebpf.io/linux/concepts/resource-limit/

[118] Linux, "Bpf verifier limits and constraints," 2024. [Online]. Available: https://www.kernel.org/doc/html/v6.9/bpf/bpf_design_QA.html#q-what-are-the-verifier-limits

[119] E. Authors, "The past, present, and future of ebpf and its path to revolutionizing systems," 2024, accessed: 2024-10-22. [Online]. Available: https://eunomia.dev/blogs/ten-years/#memory-management-upgrades-dynamic-stacks-and-more

[120] G. C. Necula, J. Condit, M. Harren, S. McPeak, and W. Weimer, "CCured: Type-safe retrofitting of legacy software," *ACM Trans. Program. Lang. Syst.*, 2005.

[121] K. Huang, M. Payer, Z. Qian, J. Sampson, G. Tan, and T. Jaeger, "Comprehensive memory safety validation: An alternative approach to memory safety," *IEEE Security & Privacy*, vol. 22, no. 4, 2024.

[122] K. Huang, Y. Huang, M. Payer, Z. Qian, J. Sampson, G. Tan, and T. Jaeger, "The taming of the stack: Isolating stack data from memory errors," in *Proceedings of NDSS*, 2022.

[123] K. Huang, M. Payer, Z. Qian, J. Sampson, G. Tan, and T. Jaeger, "Top of the heap: Efficient memory error protection of safe heap objects," in *Proceedings of the 2024 ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*, 2024.

[124] J. Devlieghere, "Escape analysis and capture tracking in llvm," 2017. [Online]. Available: https://jonasdevlieghere.com/post/escape-analysis-capture-tracking-in-llvm/

[125] B. Gregg, *BPF Performance Tools: Linux Systems and Applications*. Addison-Wesley Professional, 2019.

[126] D. Alden, "A proposal for shared memory in bpf programs," 2024. [Online]. Available: https://lwn.net/Articles/961941/

[127] Bootlin, "Linux kernel samples for ebpf, version 6.7," 2024, accessed: 2024-10-22. [Online]. Available: https://elixir.bootlin.com/linux/v6.7/source/samples/bpf

[128] IOVisor, "Bcc: Bpf compiler collection," 2024, accessed: 2024-10-22. [Online]. Available: https://github.com/iovisor/bcc

[129] BlackBerry, "Reverse engineering ebpfkit rootkit with blackberry's free ida processor tool," 2021. [Online]. Available: https://blogs.blackberry.com/en/2021/12/reverse-engineering-ebpfkit-rootkit-with-blackberrys-free-ida-processor-tool

[130] Linux Kernel Documentation, "Seccomp bpf (secure computing with filters)," 2019, accessed: 2024-08-31. [Online]. Available: https://www.kernel.org/doc/html/v4.19/userspace-api/seccomp_filter.html

[131] RustProject, "Rustc development guide: Compiler analyses and type system," 2024. [Online]. Available: https://rustc-dev-guide.rust-lang.org/part-4-intro.html

[132] ——, "Understanding ownership," 2024. [Online]. Available: https://doc.rust-lang.org/book/

[133] ——, "References and borrowing," 2024, accessed: 2024-10-22. [Online]. Available: https://doc.rust-lang.org/book/ch04-02-references-and-borrowing.html

[134] ——, "Lifetime syntax," 2024. [Online]. Available: https://doc.rust-lang.org/book/ch10-03-lifetime-syntax.html

[135] G. Li, H. Zhang, J. Zhou, W. Shen, Y. Sui, and Z. Qian, "A hybrid alias analysis and its application to global variable protection in the linux kernel," in *32nd USENIX Security Symposium*, 2023.

[136] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, 2019.

[137] T. Xia, H. Hu, and D. Wu, "DEEPTYPE: Refining indirect call targets with strong multi-layer type analysis," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[138] A. S. Elliott, A. Ruef, M. Hicks, and D. Tarditi, "Checked C: Making C safe by extension," in *IEEE Cybersecurity Development*, 2018.

[139] J. Zhou, J. Criswell, and M. Hicks, "Fat pointers for temporal memory safety of c," *ACM Program. Lang.*, no. OOPSLA1, 2023.

[140] A. Khan, D. Xu, and D. J. Tian, "Low-cost privilege separation with compile time compartmentalization for embedded systems," in *2023 IEEE Symposium on Security and Privacy*, 2023.

[141] ——, "Ec: Embedded systems compartmentalization via intra-kernel isolation," in *2023 IEEE Symposium on Security and Privacy*, 2023.

[142] Y. Zhai, Z. Qian, C. Song, M. Sridharan, T. Jaeger, P. Yu, and S. V. Krishnamurthy, "Don't waste my efforts: Pruning redundant sanitizer checks by Developer-Implemented type checks," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024.

[143] A. Limited, *Arm Memory Tagging Extension Whitepaper*, Arm Limited, 2019, accessed: 2024-08-31. [Online]. Available: https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/Arm_Memory_Tagging_Extension_Whitepaper.pdf

[144] R. N. M. Watson, S. W. Moore, P. Sewell, B. Davis, and P. Neumann, *Capability Hardware Enhanced RISC Instructions*, 2019. [Online]. Available: https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/

[145] aya, "aya: A modern, safe, and extensible ebpf library for rust," https://github.com/aya-rs/aya, 2021.

Appendix A.
Meta-Review

The following meta-review was prepared by the program committee for the 2025 IEEE Symposium on Security and Privacy (S&P) as part of the review process as detailed in the call for papers.

A.1. Summary

This paper conducts a systematic analysis of memory safety risks to the Linux kernel's eBPF runtime. The analysis shows that 1.62-3.74% of memory operations in public eBPF programs are not defended by existing defenses that include isolation mechanisms, runtime checks, or static validation. The paper proposes future research directions to close this gap while preserving eBPF's current level of functionality and performance.

A.2. Scientific Contributions

- Provides a Valuable Step Forward in an Established Field
- Independent Confirmation of Important Results with Limited Prior Research

A.3. Reasons for Acceptance

1) The paper systematizes the current state of eBPF attacks and defenses, serving as a valuable resource for researchers seeking to improve the security of eBPF.
2) The paper highlights specific gaps in current defenses based on a study of real eBPF programs, which should assist the research community in focusing their efforts on operationally relevant problems in this space.