Spinner: Detecting Locking Violations in the eBPF Runtime

Priya Govindasamy
University of California, Irvine

Joseph Bursey
University of California, Irvine

Hsin-Wei Hung
Meta

Ardalan Amiri Sani
University of California, Irvine

***Abstract*—The eBPF technology is widely used for many applications, including tracing, packet filtering, network usage monitoring, and so on. The versatility of eBPF allows the kernel's capabilities to be extended without needing to modify source code or load kernel modules. However, the eBPF subsystem may introduce new bugs that could lead to crashes, data loss, and other issues that can negatively impact system stability, reliability, availability, security, and overall performance. Specifically, locking violations, which occur when locks are not used correctly, can lead to problems like deadlocks and system hangs. Since eBPF operates at the kernel level, errors here have far-reaching consequences.**

To tackle this issue, we present Spinner, a tool for detecting locking violations in the eBPF runtime. Spinner uses static analysis to (1) detect cases of context confusion where incorrect locking primitives are used in eBPF helper functions given their execution context, and (2) identify locks in helper functions that can be called recursively using nested eBPF programs. Both of these situations could result in deadlocks. So far, Spinner has identified 34 locking violation bugs in the eBPF subsystem in Linux, only 5 of which were previously found by Syzbot.

***Index Terms*—eBPF, locking violations, static analysis, bug detection**

I. Introduction

eBPF (Extended Berkeley Packet Filter) [1] is a technology that can run programs in a privileged context, such as within the Linux kernel [2]. With eBPF, it becomes possible to dynamically add executable bytecode inside the Linux kernel to tune its behavior without having to recompile the kernel or use kernel modules. eBPF is now widely used in the Linux kernel for several applications. Ensuring the reliability of the eBPF subsystem is necessary to prevent performance issues and system crashes. Therefore, detecting bugs before they cause failures is essential to ensuring safe and efficient use of eBPF in production environments.

There have been many attempts to find bugs and vulnerabilities in the eBPF subsystem. Most previous solutions, however, focus on the eBPF verifier [3], [4], [5], [6]. While important, we note that the *eBPF runtime, i.e., the components of eBPF that execute a program after it passes the verifier*, also has a large footprint in the kernel. Hence, it is critical to analyze the eBPF runtime as well in order to find bugs and vulnerabilities.

Kernel fuzzers, such as Syzbot [7], are potentially capable of finding bugs in the eBPF runtime. They, however, face a difficult challenge in that the eBPF verifier rejects many of their inputs, lowering their coverage. BRF [8] is a kernel fuzzer that focuses on finding bugs in the eBPF runtime. It tries to fuzz the runtime by generating fuzzing inputs that successfully pass the verifier. Fuzzing, however, is a best-effort approach and could have many false negatives. Overall, BRF is reported to have found 6 bugs and vulnerabilities.

In this paper, we take a different approach. By studying the eBPF runtime, we identify a prevalent problem: *locking violations*, which result in deadlocks. We identify two underlying reasons that could result in locking violations in the eBPF runtime. We then introduce our tool, Spinner[1], which uses static analysis to find a large number of such violations in the eBPF runtime. Our use of static analysis allows us to bypass the verifier and achieve high coverage in the eBPF runtime.

We refer to the first reason underlying locking violations in the eBPF runtime as *context confusion*. Since eBPF programs are used for a wide variety of purposes, they are capable of running in varied execution contexts in the kernel. Different execution contexts require different locking mechanisms to protect shared objects. Our observation is that since different eBPF programs can execute in varied execution contexts, implementing safe locking in the eBPF runtime is error-prone. Confusing the context in which code can be executed can lead to locking violations and deadlocks.

We refer to the second reason underlying eBPF locking violations as *nested locking*. eBPF programs can be nested; the execution of an eBPF program can trigger the execution of another eBPF program. Such programs can lead to self-deadlocking issues since the thread executing these programs might try to take the same lock twice.

We have addressed several challenges in Spinner. One challenge is determining all the execution contexts that one eBPF program type can run under. To do this, we rely on eBPF self-tests [9], which contain several eBPF programs of different types. Spinner runs these programs to gather execution context information. Another challenge is determining the functions within the kernel (i.e., helper functions and kfuncs) that each eBPF program type can access. For helper functions, we use the eBPF verifier, which needs to have this information to perform safety checks. For kfuncs, we find this information in the source code.

Another important challenge with the use of static analysis is that it produces false positive reports. Manually testing these reports to determine if they are real bugs can be an arduous process. To mitigate this issue, we automate the testing of

[1]The open source implementation of Spinner is available at https://github.com/trusslab/spinner

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00180

reports using symbolic execution with S^2E [10]. That is, we use S^2E to try to generate a proof of concept (PoC) test case that can trigger the bug reported by Spinner's static analysis, hence verifying the report. To make this feasible, we automatically generate eBPF programs that are capable of testing the helper function of interest while passing the verifier's checks. We also carefully decide which values to make symbolic during the testing. Choosing to make too many values symbolic can lead to getting stuck or path explosion, and too much time spent exploring uninteresting paths, while making too few values symbolic can prevent interesting paths from being explored.

We have used Spinner to analyze the Linux kernel. It has so far found 34 locking violations. We have reported many of these violations to the Linux community already [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], and we are in the process of reporting the rest. Moreover, Spinner can automatically generate PoCs for 24 of these bugs. Interestingly, only 5 of these 34 violations were previously found by Syzbot, the state-of-the-art kernel fuzzer from Google. This shows the effectiveness of Spinner, given that Syzbot has been fuzzing the kernel 24/7 for about 7 years now.

We initially used Spinner to analyze Linux-6.9 and found 28 bugs. A key question is whether Spinner will be useful for detecting new locking violation bugs in the future. In other words, are new locking violations being introduced in the eBPF runtime, which could be detected by Spinner? To answer this question, we later analyzed Linux-6.13 as well. Spinner found 6 new bugs in Linux-6.13. This demonstrates the usefulness of Spinner for testing the Linux kernel as it is continually updated.

Finally, our experience with reporting these bugs to the Linux community has been that patching them faces some important challenges. We discuss some of these challenges in this paper as well, hoping that it will lead to better solutions.

II. Background

A. eBPF

eBPF is a powerful and flexible technology that allows users to safely run custom programs within the Linux kernel, without having to recompile the source code or use kernel modules. Although it was originally developed as a tool to filter packets, it has evolved into a general-purpose mechanism to provide many other functionalities, including monitoring and tracing system events, performance tuning, and enhancing system security.

In order to accommodate all these functionalities, there are several different *eBPF program types* [22]. Currently (as of v6.16), there are 33 program types defined in Linux that specify the intended use of the program. Some examples of these program types include kprobe [23], traffic control [24], and perf event [25]. Additionally, eBPF programs can be associated with *attach points/hook points* [26] which determine where and when an eBPF program will be invoked.

The program type also determines which eBPF *helper functions* [27] will be available to the program. Helper functions are functions within the kernel that are specifically designed to be called by eBPF programs. They are used by eBPF programs to interact with the system and perform specific tasks. For example, eBPF helper functions can be used to print debugging messages, to get the process PID, or to manipulate network packets.

Importantly, helper functions are also used to interact with *eBPF maps* [28]. eBPF maps provide programs with persistent storage. They allow storing and retrieving information as well as interaction between multiple eBPF programs and the user space. eBPF maps can be accessed from both eBPF programs and the user space (via syscalls). Several map types are available, including hash tables, arrays, ring buffers, stack trace, least-recently-used, longest prefix match, and so on.

eBPF programs can also make use of *kfuncs (Kernel Functions)* [29]. Kfuncs are also functions in the Linux kernel that are exposed for use by eBPF programs. Similar to helper functions, they allow eBPF programs to extend their capabilities and perform certain tasks. The key difference between kfuncs and helper functions is that helper functions are designed solely for use by the eBPF subsystem, while kfuncs may serve other kernel subsystems. Kfuncs are specially annotated to be callable from eBPF programs. Unlike eBPF helpers, kfuncs do not have a stable interface and can change from one kernel release to another.

B. Execution Contexts and Choice of Locks

Kernel code runs either in *process context* or *interrupt context*. Process context refers to the execution state of a process when it is running in the kernel, e.g., when issuing a syscall. Interrupt context refers to the execution in the kernel when it handles events like those triggered by hardware devices or timers. Interrupt context can be further partitioned into three categories: top and bottom halves, and Non-Maskable Interrupt (NMI). Top halves include work that must be performed on a time-critical basis. In Linux, this execution context is referred to as *hardIRQ*. Bottom halves consist of work that can be deferred to a later time, a context referred to as *softIRQ*. In addition to these, there is a special type of interrupt called NMI. Unlike regular interrupts, which can be disabled or delayed, NMIs are used for high-priority events that require immediate attention from the system, regardless of what the CPU is currently doing.

Therefore, overall, there are four execution contexts a kernel thread can execute in: process, softIRQ, hardIRQ, and NMI. A hierarchy exists between these contexts in the above order in the sense that threads that execute with contexts higher in this order can preempt those running with contexts lower in the order, but not vice versa. For example, a softIRQ can be preempted by a hardIRQ but not the other way around.

The execution context of code directly impacts the choice of locking mechanisms used within it [30], [31]. Different locking primitives are available to protect data structures that can be accessed in different execution contexts. For example, a data structure that can be accessed in hardIRQ context needs to be protected by a lock that disables hardIRQs. This

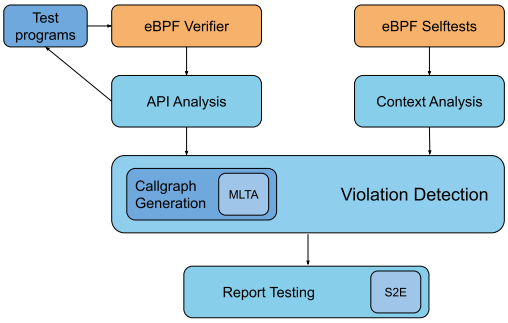

Fig. 1. Architecture of Spinner.

will prevent other hardIRQs from interrupting the currently executing thread while it is trying to access the data structure, thus preventing concurrency bugs. Therefore, there are different variants of locking primitives that are suitable for different execution contexts. In other words, the type of lock chosen to protect shared data structures often depends on the highest context in which the data structure is intended to be accessed. For example, using `spin_lock` to protect an object that could be accessed in softIRQ context is not sufficient or safe [32]. Consider the situation where a process takes the `spin_lock` and is interrupted by a softIRQ that tries to take the same lock. The softIRQ will be unable to get the lock since it has already been taken. But the process cannot complete its execution and release the lock since it cannot interrupt the softIRQ. This causes a deadlock. In this case, `spin_lock_bh` should be used. Thus, if the context for the program is not identified correctly, it could lead to deadlocks.

III. Design

We develop Spinner to detect locking violations in the eBPF runtime. We focus on the eBPF runtime since eBPF programs (and hence the helper functions they call) can be executed in various execution contexts in the kernel. Moreover, eBPF programs can be nested, which can allow helper functions to be called recursively. Finally, some helper functions can also be called from the user space. All of these factors make it challenging to avoid locking violations.

Spinner looks for two types of bugs. The first is *context confusion* bugs, which happen when incorrect locking primitives are used in the eBPF helper functions given the contexts that these helper functions can execute in. The second is *nested locking* bugs, which happen when the same thread tries to capture the same lock in an eBPF helper function that it has already acquired.

The overall architecture of Spinner is shown in Figure 1. Spinner comprises of 4 phases: API Analysis, Context Analysis, Violation Detection, and Report Testing.

Spinner begins with some knowledge about the eBPF API, including the enumerated program types and attach points, the helper function definitions, the kfunc definitions, and the enumerated map types. With this starting information, Spinner moves to the API Analysis Phase, where it determines which program types are able to call which helper functions. It accomplishes this by leveraging the eBPF verifier. It creates an eBPF program of a specific type that calls a specific helper function. Spinner then tries to load this test program into the kernel. Based on the verifier's output, it determines whether or not the helper function can be used by that program type. For kfuncs, we manually obtain this information from the source code. With this method, Spinner builds up a database of which helpers can be called by which program types.

As a direct result of the helper function analysis, Spinner can reason about what map types are available to each program type. If a program type can call a helper function that accesses or modifies maps of a specific type, then that program type has access to the map type.

Spinner then moves to the Context Analysis Phase. During this phase, it determines the possible execution contexts of each program type. It determines the execution context by running a program of a given type (using kernel self-tests) and parsing the eBPF trace logs to determine the context it was in.

Next, Spinner enters the Violation Detection Phase. It uses MLTA [33] to generate a call-graph of the Linux kernel. It then uses this call-graph as well as the results of the API analysis Phase and the Context Analysis Phase to (1) reason about the context in which helper functions can be called in order to detect context confusion violations, and (2) find scenarios when the same lock can be acquired in a nested fashion in order to detect nested locking violations. In the next two sections, we provide more details on how Spinner detects each of these two types of violations.

Finally, in the Report Testing phase, Spinner takes the reports generated in the previous phase and automatically tests them to determine whether they are real bugs. Spinner uses S^{2}E's symbolic execution to do this. This is discussed in detail in §VI.

IV. Context Confusion Violations

Context confusion violations take place when incorrect locking primitives (given the execution context) are used in an eBPF helper function, resulting in a deadlock.

A. Motivating example.

Listing 1 shows a bug [34], previously found by BRF [8], in helper functions for manipulating `BPF_MAP_TYPE_QUEUE` map.

```
1  static int __queue_map_get(struct bpf_map *map,
       void *value, bool delete)
2  {
3    ...
4
5    raw_spin_lock_irqsave(&qs->lock, flags);
6
7    ...
8
9    raw_spin_unlock_irqrestore(&qs->lock, flags);
10
11   ...
12 }
```

Listing 1. Function called by helpers to peek and pop elements in a queue map

The function `__queue_map_get` is called by the helper functions `queue_map_pop_elem` and `queue_map_peek_elem`. These helper functions can be accessed by syscalls (which execute in process context) as well as eBPF programs that could potentially execute in higher contexts. In particular, these helpers can be called in NMI context by eBPF programs of certain types, like perf events. In this case, the chosen lock is not appropriate for the contexts that can access it. Even though `raw_spin_lock_irqsave` disables interrupts while the lock is held, NMIs cannot be disabled in this way. This situation leads to a deadlock if a thread running in process context grabs the lock and is then interrupted by a thread running in NMI context. This bug has since been fixed by using a trylock instead of a spinlock if the thread is executing in NMI context.

B. Locking Rules

In order to systematically find context confusion violations, we first identify 3 locking rules needed for the correct use of locks in different execution contexts in the kernel.

There are 4 main variants of spinlock, and each has a specific use case. It is important to choose the correct spinlock variant considering the different contexts in which the shared data to be locked could be accessed. `spin_lock` is meant to be used when the data is shared only between threads that execute in process context. `spin_lock_bh` is meant to be used when the data can be accessed by threads executing in softIRQ or process context. Similarly, `spin_lock_irq` and `spin_lock_irqsave` are intended to be used when shared data can be accessed by processes running in hardIRQ, softIRQ, or process context. The main difference between these different variants is that `spin_lock_bh` disables softIRQs while `spin_lock_irq` and `spin_lock_irqsave` disable softIRQs and hardIRQs. Thus, we observe the following three rules:

Rule1. `spin_lock` should not be used by threads executing in softIRQ, hardIRQ (without disabling interrupts), and NMI contexts.
Rule2. `spin_lock_bh` should not be used by threads executing in hardIRQ (without disabling interrupts) and NMI contexts.
Rule3. `spin_lock_irqsave` and `spin_lock_irq` should not be used in NMI context.

Note that a possible safe way to use locks when the shared data can be accessed in NMI is to use *trylock*. When the thread tries to acquire the lock, if the lock is free, it can immediately acquire it. If not, the process can exit without causing a deadlock since it is not forced to spin waiting for the lock to become free.

Local locks are used to protect per-CPU data structures. Similar to the rules above, we also derive rules for them. Spinner has support to find potential bugs due to improper usage of local locks. However, we were unable to verify any of these reports automatically. Because of this and space constraints, we omit further discussion of these locks.

C. Detection Strategy

To identify context confusion violations, Spinner performs the following steps: (1) identifying eBPF program capabilities (i.e., determining which helper functions, kfuncs, and maps are available to each program type), (2) determining the execution contexts of different program types, and (3) detecting locking violations. We next elaborate on each of these steps.

1) Identifying eBPF Program Capabilities

Program types. As discussed in §II-A, there are currently 33 distinct types of eBPF programs. The type of the program restricts its functionality since a different set of helper functions is available to programs of different types.

Program types may additionally have different attach points that further specify their intended functionality. For example, a `cgroup_skb` type program used to perform packet filtering functions can be attached to hook points for ingress and egress, to work on packets that are inbound and outbound respectively.

Fortunately, this information is easily available in the kernel source code, all available in the `bpf.h` header file. Therefore, we extract the information from the source code directly. This step is performed manually as it is a short and straightforward process.

Helper Functions. Next, we need to determine which helper functions each program type can call. Although this information is present in the source code, it is highly dispersed and difficult to extract. Additionally, while the official documentation for the eBPF subsystem also contains this information, they do not account for specific attach points.

Our solution is to use the eBPF verifier. Before an eBPF program can be loaded into the kernel, it is checked by the eBPF verifier to make sure that the program will not crash, hang, or otherwise interfere with the kernel negatively. One of the checks performed by the verifier is whether the helper functions invoked by the program are allowed for programs of its type. This ensures that eBPF programs can only access a selected set of functionality.

We make use of the verifier to determine whether a particular program type can use a particular helper function. We first extract the list of all available helper functions from the kernel source code (from the aforementioned header file). We then generate an eBPF program that tests a different helper function in each run. Every helper function is tested against every program type. We also develop a program in C and use it to load and attach the generated eBPF programs.

The generated eBPF test program uses an ELF section name to specify its type and attach point. We use the Libbpf API [35] to load and attach the generated test programs. Libbpf is a C-based library used for preparing and loading compiled bpf object files into the Linux kernel. Libbpf takes care of loading, verifying, and attaching eBPF programs to the various attach points in the kernel. To generate programs of different types, Libbpf provides a convenient API that allows programmers to

specify an ELF section name for an eBPF program to indicate its type and attach point.

We try to load the test program into the kernel. If the helper function used by the test program is not available for that particular program type, the verifier will throw certain types of errors and prevent the program from being loaded. On the other hand, if the test program passes those verifier checks, it indicates that the helper function is allowed to be used by that program type. In this way, we test all the program types and helper functions and create a comprehensive mapping of helper functions to program types.

Kfuncs. Next, we need to determine which kfuncs each program type can call. The kfuncs that can be used by an eBPF program depend on its program type. In the source code of the Linux kernel, where the kfuncs are defined, they need to be registered with the program types that are allowed to use them. Consider the example in Listing 2. Here, two kfuncs are defined and annotated. These kfuncs are used for manipulating bpf arena, which is a region of memory that can be shared between eBPF programs and user space programs. Lines 10 to 13 show that these two kfuncs are grouped together as `arena_kfuncs`. Line 17 shows the annotation for registering a set of kfuncs with a program type. Here, the program type is `BPF_PROG_TYPE_UNSPEC`, which indicates that these kfuncs are available to programs of all types. In this way, we extract the information from the kernel source code. This step can be easily automated, although we have not done so yet.

```
1  __bpf_kfunc_start_defs();
2
3  __bpf_kfunc void *bpf_arena_alloc_pages (void
       *p__map, void *addr__ign, u32 page_cnt, int
       node_id, u64 flags)
4  { ... }
5
6  __bpf_kfunc void bpf_arena_free_pages (void
       *p__map, void *ptr__ign, u32 page_cnt)
7  { ... }
8  __bpf_kfunc_end_defs();
9
10 static const struct btf_kfunc_id_set
       common_kfunc_set = {
11   .owner = THIS_MODULE,
12   .set   = &arena_kfuncs,
13 };
14
15 static int __init kfunc_init(void)
16 {
17   return register_btf_kfunc_id_set
       (BPF_PROG_TYPE_UNSPEC, &common_kfunc_set);
18 }
19 late_initcall(kfunc_init);
```

Listing 2. Definition and annotations for arena kfuncs

Map types. Next, we need to determine the map types that each program type can use. eBPF maps are used to provide storage of different types and to share data between kernel and user space programs. There are currently 36 different map types available, including hash, array, bloom filter, and cgroup storage.

Maps such as array maps and hash maps can be accessed by any program type using generic helper functions `bpf_map_update_elem`, `bpf_map_lookup_elem`, and `bpf_map_delete_elem`. Internally, these functions have different implementations based on the map that is being accessed.

```
#                              _-----=> irqs-off/BH-disabled
#                             / _----=> need-resched
#                            | / _---=> hardirq/softirq
#                            || / _--=> preempt-depth
#                            ||| / _-=> migrate-disable
#                            |||| /     delay
#           TASK-PID     CPU#  |||||  TIMESTAMP  FUNCTION
#              | |         |   |||||     |         |
 systemd-journal-492     [012] ...21 79280.104327: bpf_trace_printk: BPF

   kworker/u76:2-7636    [011] ...21 79280.902871: bpf_trace_printk: BPF
```

Fig. 2. Format of eBPF logs.

Several map types exist to support specific BPF helpers that perform actions based on the map contents. As such, these maps can be accessed from eBPF programs only through specific eBPF helper functions. The program types that can access these helpers are identified using the procedure in §IV-C1. If a program of a particular program type can access a helper function to manipulate a map of a specific map type, it follows that maps of that type are available to programs of that type. For example, to access maps of the type `BPF_MAP_TYPE_REUSEPORT_SOCKARRAY`, the helper function `bpf_sk_select_reuseport` is used. This helper function is available only to programs of type sk_reuseport. Thus, maps of type `BPF_MAP_TYPE_REUSEPORT_SOCKARRAY` are accessible only to programs of type `BPF_SK_REUSEPORT`. Maps that are accessed only through the generic API are identified manually.

2) Determining Execution Context

Each eBPF program type can run in a specific subset of all possible execution contexts. Each program type has one or more attach points/hook points to which these programs can be attached. Pre-defined hooks include syscalls, function entry/exit, kernel tracepoints, network events, and several others. *An eBPF program is executed when the kernel or an application passes the hook point to which it is attached. The hook point of an eBPF program determines its execution context.*

We perform dynamic analysis to determine the execution contexts of different program types. We use the helper functions `bpf_trace_printk` and `bpf_printk` that are used by eBPF programs to print debugging messages. These functions are available to programs of all program types. The output of these messages have the same format as ftrace [36] logs. This means they include other useful information such as the task name, PID, and the CPU running the program. Importantly, it includes information about the execution context the program is running in. We make use of this fact to determine the execution context of the different program types.

To achieve high coverage, we use the extensive set of eBPF programs readily available in the kernel self-tests. A print statement is inserted into every eBPF program that prints out its ELF section name. This enables the identification of its program type and attach point. Once the self-tests are run, the results can be extracted from the logs. The format of the

TABLE I
SYMBOLS USED TO REPRESENT EXECUTION CONTEXT IN TRACE LOGS AND THEIR MEANINGS

Symbol	Meaning
Z	NMI occurred inside a hardIRQ
z	NMI is running
H	HardIRQ occurred inside a softIRQ
h	HardIRQ is running
s	SoftIRQ is running
.	Process context

logs is shown in Figure 2. The important thing here is that the logs include context information in the field hardIRQ/softIRQ. They are represented by the symbols in Table I.

Please note that while Spinner mainly relies on static analysis for its bug detection, it uses dynamic analysis to determine the possible execution contexts of each program type. This is because we could not find a reliable static analysis approach to derive the execution context without a significant amount of false positives and negatives.

The dynamic analysis produced results consistent with expectations from manual analysis. Unfortunately, we cannot do better to evaluate this than manual analysis, since we have no way to establish the ground truth. We note that since we may not achieve complete coverage, we may miss some bugs. But we use dynamic analysis as part of a best effort approach and try to find as many bugs as possible.

3) Detecting Violations

As a result of the introduced analyses, Spinner now knows the execution context of the eBPF program types as well as which helpers can be used by which program types. Each helper can be called by one or more program types. The execution contexts of these program types can be compared to find the highest context in which that helper function can be called. For example, the helper function `bpf_csum_level` can be used by the program types `SCHED_CLS` and `LWT_IN`, among others. `SCHED_CLS` programs execute in softIRQ context while `LWT_IN` programs execute in process context. Here we find that the highest context this helper function can be called in is softIRQ. We perform similar analyses for all the helper functions and kfuncs, thus identifying their highest execution contexts.

Spinner then uses the call-graph of the Linux kernel to identify violations of the rules mentioned in §IV-B. To generate the call-graph, it uses static analysis using the Multi-Layer Type Analysis(MLTA) [33]. Spinner then traverses the call-graph using a depth-first search to determine the set of functions that are called by eBPF helper functions. It uses the previously gathered information on the highest execution context of a helper function or kfunc, the list of functions invoked by each helper function or kfunc, to find any functions that violate the locking rules. For instance, consider the helper function `bpf_csum_level`. The highest context it can execute in is softIRQ. Therefore, this function must disable bottom halves before taking any locks, so using only `spin_lock` is not allowed. Such checks are performed for all helper functions and kfuncs. For example, for **Rule1**, Spinner searches for instances of `spin_lock` or `raw_spin_lock` in functions whose highest contexts are softIRQ, hardIRQ, or NMI.

V. NESTED LOCKING VIOLATIONS

Nested eBPF refers to the concept where one eBPF program invokes another eBPF program. This occurs when the execution of one eBPF program may involve executing the hook point to which another eBPF program is attached, thus triggering the execution of the second eBPF program.

There are certain situations in which nested eBPF may lead to a deadlock. One such situation is *recursive eBPF*, wherein a program could trigger itself recursively. In such cases, trying to take the same lock would lead to a deadlock. To prevent recursive issues, eBPF program recursion prevention is implemented in the kernel. This is done in two ways. Firstly, some eBPF program types are restricted such that programs of these types are prevented from executing when another program of a restricted type is active. This applies to kprobe, tracepoint, and perf event programs. Secondly, for some program types, the same program cannot be executed when one instance of it is already executing. This applies to fentry, fexit, tracepoint, raw_tracepoint, and raw_tracepoint_ writable programs. All the program types for which recursion prevention is implemented share the property that they can be hooked to arbitrary locations in the kernel and are called at arbitrary times. For instance, kprobe programs can be hooked to nearly every instruction. Hereafter, we refer to these program types as *tracing* (and refer to other program types as non-tracing).

Note that although eBPF program recursion prevention measures can prevent some of the deadlocks caused by nested eBPF, we find that it does not prevent all possible cases.

Motivating example. Consider the example in Figure 3 where two eBPF programs try to manipulate the elements of a sock map. A sock map can be used to redirect socket buffers between sockets or to apply policy at the socket level based on the result of an eBPF program.

There are two eBPF programs in this example, the first one is a traffic control program ① that is executed whenever there are packets to be processed. The other is a tracepoint program that is triggered whenever `kfree` is called. The traffic control eBPF program attempts to delete elements of a sock map using the map helper function `sock_map_delete_elem` ②. This helper function takes a `spin_lock_bh` ③, but before it can release the lock, `kfree` is called to free up some memory④. At this point, the execution of the traffic control program is paused, and the BPF program which is attached to the tracepoint for `kfree` will be triggered⑤. When the tracepoint program tries to update the same sock map⑥, it will try to take the spinlock⑦. However, the traffic control program has already taken the lock and cannot release it, since it cannot resume its execution until the tracepoint program finishes. Thus, the tracepoint program spins forever, leading to a deadlock. This bug [37] was found by syzbot and has since been fixed by preventing tracepoint programs from accessing sock maps.

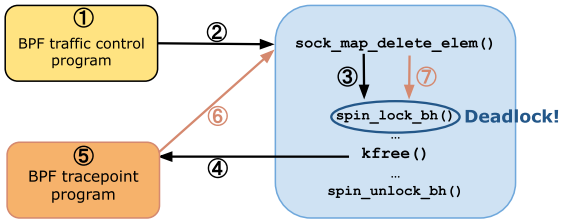

Fig. 3. Deadlock in nested eBPF. The black arrows show the execution path of the traffic control BPF program, while the orange arrows show the execution path of the tracepoint BPF program.

TABLE II
TIME TAKEN TO PERFORM DIFFERENT STEPS OF THE ANALYSIS

Component	Time taken (minutes)
API Analysis	266.50
Context Analysis	86.37
Callgraph Generation	35.90
Violation Detection	15.75
Total	**404.52**

A. Violation Patterns & Detection Strategy

We discover that nested locking violations could occur under two circumstances, leading to deadlocks.

The first circumstance is when helper functions that can be accessed by both tracing type programs and non-tracing type programs use locks. The example discussed earlier (Figure 3) is an example of this type.

The second circumstance is when helper functions that use locks can be accessed by both tracing-type eBPF programs and can be accessed from the user space. More specifically, certain helper functions can be called from the user space. These are map helper functions that are used to update, delete, and lookup elements from maps. Deadlocks can occur if these functions use locks and are accessed by both tracing-type eBPF programs and from the user space.

To identify these types of nested locking issues, Spinner performs the following steps: (1) identify helper functions that can be accessed by both tracing type programs and non-tracing type programs, (2) identify helper functions used to manipulate maps that can be accessed by tracing type programs and user space programs, and (3) check if locks are used in the helper functions identified in the previous two steps.

VI. AUTOMATED GENERATION OF PROOF OF CONCEPTS

Due to the use of static analysis, Spinner could generate false reports. To mitigate this, Spinner tries to generate Proof of Concepts (PoCs) for each report, i.e., programs that can trigger the reported bugs. If successful, the PoC can be used to demonstrate the presence of the detected bug. This way, Spinner can prune false reports. This reduces the manual effort needed to verify bugs.

At a high level, Spinner creates eBPF programs that exercise the helper function where a bug was reported. Spinner then uses S^2E [10] to make arguments to the helper functions symbolic. Using symbolic execution allows us to follow several different paths of execution for the helper function and test whether they trigger the reported bugs. After symbolic execution completes, S^2E generates concrete inputs for the helper function arguments that exercise the different execution paths of the helper function. If we find some inputs that lead to the detection of a deadlock during the symbolic execution, those inputs, along with the sample programs created by Spinner, form the PoC.

For context confusion bugs, Spinner generates two eBPF programs. The first one calls the reported helper function in a higher context than can be protected by the reported lock. The second one calls the helper function in some other lower context. During symbolic execution, the kernel's locking validator, Lockdep [38], is also active. If Lockdep detects that the reported lock could be taken by programs in both contexts in an unsafe way, it prints out a report.

For nested locking bugs, Spinner again generates two eBPF programs. The first one calls the reported helper function normally. The second one also calls the reported helper function, but it tries to attach to a point in the reported lock's critical section using kprobe, tracepoint, or fentry programs. During symbolic execution, if a nested locking bug report is a true positive, deadlock can occur. Although this proves the existence of a deadlock bug, running into a deadlock can prevent S^2E from completing symbolic execution and generating test cases. To prevent this from happening, we created an S^2E plugin that detects self-deadlocks and terminates paths that self-deadlock.

To ensure successful symbolic execution of the sample eBPF programs, Spinner uses a template-based approach. The sample eBPF program generated by Spinner must be semantically correct. Otherwise, the verifier will refuse to load it, and symbolic execution cannot proceed. Many helper functions and kfuncs have their own specific requirements that must be fulfilled in order to ensure that verification completes successfully. Additionally, some helpers and kfuncs take parameters as inputs that can only be acquired by calling some other helper function or kfunc in prior. These types of dependencies must be modeled in the form of templates to allow correct sample program generation.

So far, we have created templates for working with generic eBPF maps like hash and queue maps. These maps are where Spinner reported the most bugs. Spinner is also able to handle helper functions and kfuncs that do not have any specific dependencies. With these templates, we are able to automatically test 76% of Spinner's reports. This also allowed the automated discovery of 24 bugs. We leave the creation of more templates and testing other reports for future work.

VII. EVALUATION

A. Execution Time

We execute Spinner on a virtual machine with 16 2GHz Intel Xeon Processor cores. We report the execution time of the analysis of the Linux-v6.9 kernel. The total time taken to execute API analysis, Context Analysis, and Violation Detection components of Spinner was 400.82 minutes. The individual breakdowns for different components of the analysis are shown in Table II. Most of the time is spent in the API analysis phase to identify which helper functions can be called

TABLE III
CONTEXT CONFUSION BUGS UNCOVERED BY SPINNER

No.	Violating Lock	Function	Affected Helpers/Kfuncs	Highest Context	Kernel Version	Syzbot found
1	spin_lock_irqsave	trie_delete_elem	trie_delete_elem	NMI	v6.9, v6.13	No
2	spin_lock_irqsave	trie_update_elem	trie_update_elem	NMI	v6.9, v6.13	No
3	_raw_spin_lock_irqsave	bpf_common_lru_pop_free	htab_lru_map_update_elem	NMI	v6.9, v6.13	No
4	_raw_spin_lock_irqsave	bpf_percpu_lru_pop_free	htab_lru_percpu_map_update_elem	NMI	v6.9, v6.13	No
5	_raw_spin_lock_irqsave	bpf_common_lru_push_free	htab_lru_map_update_elem, htab_lru_map_delete_elem	NMI	v6.9, v6.13	No
6	_raw_spin_lock_irqsave	bpf_percpu_lru_push_free	htab_lru_percpu_map_update_elem, htab_lru_percpu_map_delete_elem	NMI	v6.9, v6.13	No
7	_raw_spin_lock_irqsave	percpu_counter_sum	htab_percpu_map_update_elem	NMI	v6.9, v6.13	No
8	_raw_spin_lock	percpu_counter_add_batch	htab_percpu_map_update_elem, htab_percpu_map_delete_elem	NMI	v6.9, v6.13	No
9	raw_spin_lock	bpf_lru_list_pop_free_to_local	htab_lru_map_update_elem, htab_lru_map_delete_elem	NMI	v6.9, v6.13	No
10	raw_spin_lock_irqsave	bpf_lru_list_push_free	htab_lru_map_update_elem, htab_lru_map_delete_elem	NMI	v6.9, v6.13	No
11	raw_spin_lock_irqsave	_cgroup_rstat_cpu_lock	cgroup_rstat_updated	NMI	v6.9, v6.13	No
12	_raw_spin_lock_irqsave	lock_timer_base	trie_update_elem	NMI	v6.9	No
13	_raw_spin_lock	krc_this_cpu_lock	trie_update_elem	NMI	v6.9	No
14	_raw_spin_lock_irqsave	destroy_dsq	scx_bpf_destroy_dsq	NMI	v6.13	No
15	_raw_spin_lock_irqsave	scx_bpf_exit_bstr	scx_bpf_exit_bstr	NMI	v6.13	No
16	_raw_spin_lock_irqsave	scx_bpf_error_bstr	scx_bpf_error_bstr	NMI	v6.13	No

TABLE IV
NESTED LOCKING BUGS UNCOVERED BY SPINNER

No.	Violating Lock	Function	Affected Helpers	Program Types	Kernel version	Syzbot found
17	_raw_spin_lock_irqsave	bpf_common_lru_pop_free	htab_lru_map_update_elem	All	v6.9, v6.13	No
18	_raw_spin_lock_irqsave	bpf_percpu_lru_pop_free	htab_lru_percpu_map_update_elem	All	v6.9, v6.13	No
19	_raw_spin_lock_irqsave	bpf_common_lru_push_free	htab_lru_map_update_elem, htab_lru_map_delete_elem	All	v6.9	No
20	_raw_spin_lock_irqsave	bpf_percpu_lru_push_free	htab_percpu_lru_map_update_elem, htab_percpu_lru_map_delete_elem	All	v6.9	No
21	spin_lock_irqsave	trie_delete_elem	trie_delete_elem	All	v6.9, v6.13	Yes
22	spin_lock_irqsave	trie_update_elem	trie_update_elem	All	v6.9, v6.13	No
23	_raw_spin_lock_irqsave	__queue_map_get	queue_map_peek_elem, queue_map_pop_elem	All	v6.9, v6.13	Yes
24	_raw_spin_lock_irqsave	__stack_map_get	stack_map_peek_elem, stack_map_pop_elem	All	v6.9, v6.13	Yes
25	_raw_spin_lock_irqsave	queue_stack_map_push_elem	queue_stack_map_push_elem	All	v6.9, v6.13	Yes
26	_raw_spin_lock	___pcpu_freelist_pop	bpf_get_stackid, htab_map_update_elem, htab_percpu_map_delete_elem	All	v6.9, v6.13	No
27	_raw_spin_lock	___pcpu_freelist_push	bpf_get_stackid, htab_map_update_elem, htab_map_delete_elem, stack_map_delete_elem	All	v6.9	No
28	raw_spin_lock	bpf_lru_list_pop_free_to_local	htab_lru_map_update_elem, htab_lru_map_delete_elem	All	v6.9	No
29	raw_spin_lock_irqsave	bpf_lru_list_push_free	htab_lru_map_update_elem, htab_lru_map_delete_elem	All	v6.9	No
30	_raw_spin_lock_irqsave	__bpf_ringbuf_reserve	bpf_ringbuf_reserve, bpf_ringbuf_output, bpf_ringbuf_reserve_dynptr	All	v6.9, v6.13	Yes
31	_raw_spin_lock	krc_this_cpu_lock	trie_update_elem	All	v6.9	No
32	_raw_spin_lock_irqsave	destroy_dsq	scx_bpf_destroy_dsq	tracing struct_ops syscall	v6.13	No
33	_raw_spin_lock_irqsave	scx_bpf_exit_bstr	scx_bpf_exit_bstr	tracing struct_ops syscall	v6.13	No
34	_raw_spin_lock_irqsave	scx_bpf_error_bstr	scx_bpf_error_bstr	tracing struct_ops syscall	v6.13	No

from which program type. This portion of the analysis needs to be performed only once.

We then used Spinner's automated testing framework to test all of Spinner's reports for Linux-v6.9 and Linux-v6.13 for which we had templates. The symbolic execution used in this process may take a long time to fully test all possible paths. Due to time constraints, we test each report for 15 minutes using 48 parallel processes on the aforementioned server.

B. Discovered Bugs

Using Spinner, we have uncovered 34 bugs that could lead to deadlock. 17 of these bugs are instances of context violation, and the other 18 are instances of nested locking violation. Tables III and IV list all of these bugs.

C. Comparison with BRF and Syzkaller

To compare Spinner with BRF and Syzkaller [39], we ran BRF and Syzkaller for a period of 72 hours each, allowing each fuzzer to use 4 VMs. Each VM was configured with 2 virtual CPUs and 2GB memory. To give syzkaller a better chance of finding bugs within the eBPF subsystem, we enabled only the syscalls necessary for process and memory management, networking, and eBPF. Neither fuzzer was able to find deadlock bugs in the eBPF subsystem within the 72-hour period.

As reported in the BRF paper [8], BRF found 3 deadlock bugs in eBPF. We have been able to confirm that 2 of these bugs were due to context confusion. These bugs have been patched since then and thus were not found by Spinner.

We also investigated whether any of the bugs discovered by Spinner were previously found by the continuous kernel fuzzer Syzbot [7], given that Syzbot has been fuzzing the kernel 24/7 for about 7 years. Our investigations show that (as of 08/2025) Syzbot had only found 5 of these bugs (all of which are instances of nested locking violation) as shown in Table IV.

We suspect that many of these bugs have been in the kernel for a long time and have gone undetected. For example, we checked the Syzbot report for the 5 bugs that it found. Our investigations of the reports (and the additional information posted by the developers for each of the bugs) show that these 5 bugs were all introduced in 2017-18. They were all detected in 2024, meaning they were undetected for 6-7 years. These results show the effectiveness of Spinner in detecting locking violations in the eBPF runtime.

We additionally investigated all the bugs reported by Syzbot in the eBPF subsystem to see if Spinner failed to find any context confusion or nested locking bugs present in the kernel versions that we evaluated. We found that Syzbot found one bug in the helper function `bpf_trace_printk` [40] which Spinner did not. Spinner was not able to find this bug due to a difference in the kernel configuration, which led to the code that this bug was in not being compiled into the kernel we evaluated.

D. Effectiveness of Automated Testing

Spinner's automated testing is capable of detecting a large number of the bugs that we have identified. Overall, Spinner's automated testing framework was able to detect 24 of the 34 bugs listed in Tables III and IV. Spinner's static analysis generated a total of 1780 reports in Linux-v6.9 and Linux-v6.13. We managed to automatically test 1356 of the reports since we did not have the right template to test the rest. Automatically testing uncovered 53 of the reports to be real bugs. These 53 reports corresponded to 24 unique bugs.

We identified the remaining 10 bugs (out of 34) through manual analysis. As discussed below, we perform manual analysis of some reports for Linux-v6.9 in order to measure the accuracy of our automated testing. We also perform opportunistic manual testing of some of the reports in Linux-v6.13, where we suspect a report could be a real bug. Out of these 10 bugs, 7 of them are in reports for which we do not have the templates for automatic testing. Therefore, adding more templates can further improve automated testing in the future. 3 of these bugs, however, were among the reports that we automatically tested. This shows that our automated testing is not 100% accurate. Next, we try to measure its accuracy.

E. Accuracy of Automated Testing

In order to measure the accuracy of Spinner's automated testing framework, we need to compare the results from the automated testing against the ground truth. Obtaining the ground truth is, however, challenging as it requires manual analysis of all the reports by Spinner's static analysis. Therefore, in this section, we only focus on Linux-v6.9. Moreover, we only focus on those reports for which automated testing has a template for testing.

Determining ground truth. Spinner's static analysis generates 846 reports in Linux-v6.9 for which we have templates for automated testing. We could not analyze all of these reports one by one, as analyzing each report took us about 1-2 hours. Therefore, we first used some heuristics to detect a large number of false reports based on our domain knowledge gained from analyzing a large number of reports. More specifically, we prune reports based on two observations. Firstly, we observe that there are certain functions that have been carefully protected and are safe to call from any context. For example, functions that are meant for printing, like `vkdb_printf`, and functions meant for instrumentation, like `lock_acquire`, are safe to use in any context but may use locks which Spinner reports on. We remove this type of false positive by removing reports that include these functions. Furthermore, we also ignore reports that include functions that do not make sense to check. For instance, it does not make sense to check for locks used in the `panic` function since this would indicate that some other critical error that the operating system cannot recover from has already been triggered. We then manually analyzed the remaining 108 reports one by one.

Comparison with automated testing. The results are shown in Table V. The table shows the number of reports that lead to deadlock, the number of unique bugs found by the manual analysis (ground truth), and the number of bugs found by automated testing. These results show that Spinner's automated testing correctly detected 74% of reports that lead

TABLE V
ACCURACY OF SPINNER'S AUTOMATED TESTING FRAMEWORK BY COMPARISON TO MANUALLY IDENTIFIED GROUND TRUTH

	No. of reports that lead to deadlock	No. of unique bugs
Ground Truth	39	24
Automated Testing	29	21

to deadlocks. Moreover, it identified 87.5% of the unique bugs. Overall, these results show that our automated testing is quite accurate when it has the templates to test a report.

F. Mitigating issues in symbolic execution

We observe two major issues with symbolic execution. If too much data is made symbolic, the S^2E engine may get stuck in the constraint solver. This prevents any further progress. The other issue is path explosion. When too many paths are created, interesting paths may not be explored.

To overcome these issues, Spinner implements two mitigations. First, only selected data is made symbolic. For example, to test map helper functions, rather than making the large `struct bpf_map` symbolic, we choose to make only the map flags and maximum entries in the map symbolic. This helps to prevent getting stuck in the constraint solver. Second, Spinner allows forking, or additional paths to be created, only in code related to eBPF. This allows the symbolic execution to explore mostly interesting paths.

To evaluate the effect of these mitigations, we perform an ablation study. We select a subset of size ten of the bugs. For each bug, we perform the following four tests:

- Without any mitigations
- With selected arguments made symbolic
- With forking permitted only within the eBPF subsystem
- Spinner: With both selected arguments made symbolic and forking permitted only within the eBPF subsystem.

We run each test five times and measure the number of paths created during symbolic execution, and determine whether progress is halted due to getting stuck in the constraint solver.

TABLE VI
ABLATION STUDY

Values are the mean ± standard deviation of the number of paths created during a 15-minute run of each test. Green( ) indicates that the bug was found. Red( ) indicates that the symbolic execution engine got stuck in the constraint solver. Orange( ) indicates that the bug was not found within the allocated time, although the engine did not get stuck. Split cells show that two behaviors occurred within that test. The cell is divided in a 2:3 ratio to reflect that the first behavior was seen in two runs and the second in three runs; each segment reports its own mean ± SD.

Bug No.	No optimization	Disallow Forking	Constrain Arguments	Spinner: Both optimizations
2	3567.6±796.7	27.6±9.8	3808.6±179.2	257.2±31
22	4±0	4±0	20±11.3	2013±263.8
31	629±160 | 36.3±0.6	18±4	4681±1241.6	1913±140.4
1	4151.2±755.2	36±26	7760.8±980.2	360.6±40
21	49±9.9 | 419.3±295.6	32.8±9	991±0	187.6±39
3	61.4±10.5	12.4±7.9	899.8±199.4	82±8.2
6	59.4±35.6	3.6±1.7	1815.2±392.7	100.4±62.9
5	53.6±41.6	9.2±4.8	10.4±0.9	71.6±17.6
4	71.2±47.7	6.2±2.9	49±39.6 | 2107.3±664.9	88.2±56.9
18	13±0	3±0	1±0	1±0

The mean and standard deviation of the number of paths created in each test are reported in Table VI. As can be seen, with no optimization and when symbolic arguments are not constrained, the symbolic execution engine often gets stuck while solving constraints due to the large amount of symbolic data. And in cases where it does not come to a complete halt, interesting paths are not explored, and the bug cannot be found. With the optimization to constrain arguments, the bug can often be found, but a large number of paths are created. This can once again sometimes lead to the symbolic execution engine getting stuck due to the creation of more and more new symbolic data as the test runs. With both optimizations, we are able to find the bug in all tests.

VIII. RELATED WORK

Since static analysis examines the entire codebase, including paths that might be hard to reach during runtime, it can find potential bugs that might not be exposed through dynamic analysis. Some prior tools [41], [42], [43], [44] use simple rules to apply static analysis techniques to the whole Linux kernel. Others [45], [46], [47], [48], [49] apply these techniques to kernel modules and device drivers. And some [50], [51], [52] use symbolic analysis to test the Linux kernel.

Static analysis approaches have been used to systematically detect deadlocks in OS kernels. RacerX [53] uses locking constraint to describe the locking situation when each lock is acquired. It performs flow-sensitive and inter-procedural analysis to identify the code paths containing locking constraints from OS kernel code, and then recursively compares between two target code paths with their locking constraints to detect locking cycles as deadlocks. DLOS [54] uses a summary-based lock-usage analysis to extract the code paths containing distinct locking constraints from kernel code and a reachability-based comparison method to detect locking cycles from locking constraints.

Static analysis techniques have also been applied to eBPF. PREVAIL [55] presents the design of an alternative verifier. Agni [56] performs range-analysis on the eBPF verifier. VEP [57] uses an annotation-guided verification approach and evaluates eBPF programs both at the C level and at the bytecode level. Jitterbug [58] applies formal methods to the BPF Just-In-Time (JIT) compilers. The framework proposed by Bhat *et al.* [6] targets the range analysis pass of the verifier and presents an automated formal verification framework that finds verifier bugs. JitSynth [59] is a tool to convert eBPF bytecode into verified native RISC-V instructions.

Fuzzing the kernel is an effective technique for identifying vulnerabilities and bugs by feeding it unexpected or random inputs. Several fuzzers like [39], [60], [61], [62], [63], [64], [65] have been instrumental in exposing unexpected behavior in the kernel. Hybrid approaches like [66], [67], [68], [69], [70], [71], [72] use both static and dynamic analyses to improve the effectiveness of vulnerability detection. Concurrency bugs, caused by improper usage of synchronization mechanisms like locks, can also be identified by fuzzing, as shown by [73], [74], [75]. SyzScope [76] and KOOBE [77] analyze the impact of

bugs identified through fuzzing.

Fuzzing has been applied to identify vulnerabilities in the eBPF subsystem as well. BRF [8] is a fuzzer designed to generate test programs that satisfy the semantics required by the eBPF verifier. BVF [4] uses a modified Syzkaller to find correctness bugs in the verifier. Other fuzzers [78], [5] detect ALU range errors and logical bugs in the verifier. Sun *et al.* introduce the concept of state-embedding to dynamically test the verifier for logic bugs in [79]. BPFChecker [80] uses differential fuzzing to detect implementation flaws in eBPF runtimes. The results of these works indicate that a number of vulnerabilities are present within the eBPF subsystem. Since fuzzing and other dynamic analyses may have false negatives, our work aims to avoid false negatives by using static analysis.

BeeBox [81] and VeriFence [82] focus on preventing side-channel attacks caused by the exploitation of eBPF programs. Wiebing and Giuffrida [83] show that in spite of current mitigations, Spectre-v2 attacks are still possible using cBPF.

Chintamaneni et al. [84] investigate the impact of nested eBPF programs on the kernel stack. They find that with sufficient amount of nesting, the kernel stack can be overflowed.

IX. Discussion

A. Patching Challenges

We submitted a patch [85] to the Linux community to fix all nested locking bugs in Table IV except bugs 31-34 (which we have found more recently). Our patch modified the Makefile to prevent ftrace hooks from being used anywhere in certain files. Since eBPF's kprobe and fentry frameworks rely on ftrace internally, this prevents these nested locking bugs from being triggered. Our patch was integrated into the mainline Linux kernel in v6.13.

However, our evaluation of Linux-v6.13 shows that many of these bugs can still be triggered. Although eBPF programs cannot be attached to any point in the patched files, they can still be attached to functions in other files that the affected helper functions may call into, thus leading to the same problem. Since these helper functions could potentially reach functions in several different files, we could not effectively determine all the files that needed to be patched.

Further, we have not been able to suggest easy fixes for some of the bugs that Spinner has found. One challenge we faced was related to improper lock usage in NMI context, especially in BPF maps. Specifically, trie and hashtable maps are currently unsafe for NMI use. The easiest option (which we suggested to the Linux community) to patch these bugs would be to use `trylock` instead of a normal lock in NMI context. However, our suggestion was rejected since this type of patch would mean that code in NMI context trying to access the map may not be able to do so, and this is not acceptable. Patching these bugs in a way that ensures the functionality of these maps for all use cases is still an open problem.

B. Need for Spinner in the future

New kfuncs are being added to the Linux kernel with each new version. Helper functions may also be updated. These factors could lead to new deadlock bugs being added to the kernel. Spinner can help to detect these bugs. Our evaluation of Linux-v6.13 led to the discovery of bugs 14, 15, 16, 32, 33, and 34. These bugs were found in kfuncs that were added in v6.12. This shows that Spinner is capable of finding bugs in newer versions of the kernel and will continue to prove useful as newer functionalities are added to the eBPF subsystem.

Recently, resilient spinlocks [86] have been added to Linux, which attempt to combat the nested locking problem. These spinlocks perform internal checking for deadlocks and return an error if taking the lock might result in a deadlock. Spinner is complementary to these locks and can guide the use of these spinlocks in the kernel.

C. Generation of New Templates

Adding templates for new helper functions and kfuncs is currently manual, and is not trivial to automate since this requires understanding the dependencies among helper functions and kfuncs. There have been research efforts to automatically generate templates for syscalls needed for kernel fuzzing, e.g., SyzDescribe [87], SyzSpec [88]. We believe similar ideas can be used to generate templates for Spinner as well.

At the moment, the manual effort required to create a template includes first analyzing the dependencies of the helper function/kfunc and then encoding those programmatically. We observe that there are three main types of dependencies. First, some arguments passed to a helper function/kfunc may be obtained only from specific sources since eBPF programs are restricted in the actions they can perform and the set of functions they can use. For example, the helper function `bpf_cgrp_storage_get` requires a `struct cgroup` argument. This could be acquired using a kfunc like `bpf_cgroup_from_id`. Second, some helper functions/kfuncs should be called only after some state is modified by others. For example, `bpf_spin_unlock` should be called only after `bpf_spin_lock`. Third, some helper functions/kfuncs need to interact with bpf maps. Therefore, the required map and, in some cases, elements within the map need to be created. Once such dependencies have been analyzed, they need to be coded such that a sample program for the required helper function will have its dependencies satisfied.

X. Conclusion

We presented Spinner, a tool that uses static analysis to detect locking violations in the eBPF runtime. Spinner identifies instances of context confusion, where locking does not match the execution context of the code. It also identifies nested locking issues that occur due to the nature of eBPF, in which a process may try to grab the same lock twice. Spinner has found 34 bugs so far, only 5 of which have been previously found by Syzbot.

XI. Acknowledgments

This work was supported in part by DARPA under Agreement No. FA8750-24-2-0002. The authors thank the anonymous reviewers for their insightful feedback.

REFERENCES

[1] "ebpf documentation," https://ebpf.io/.
[2] L. Torvalds, "Linux," https://github.com/torvalds/linux.
[3] "ebpf verifier documentation," https://docs.kernel.org/bpf/verifier.html.
[4] H. Sun, Y. Xu, J. Liu, Y. Shen, N. Guan, and Y. Jiang, "Finding correctness bugs in ebpf verifier with structured and sanitized program," in *Proceedings of the Nineteenth European Conference on Computer Systems*, 2024, pp. 689–703.
[5] Y. Li, W. Niu, Y. Zhu, J. Gong, B. Li, and X. Zhang, "Fuzzing logical bugs in ebpf verifier with bound-violation indicator," in *ICC 2023-IEEE International Conference on Communications*. IEEE, 2023, pp. 753–758.
[6] S. Bhat and H. Shacham, "Formal verification of the linux kernel ebpf verifier range analysis," 2022.
[7] G. Inc., "Syzbot: Kernel fuzzing dashboard," https://syzkaller.appspot.com, 2017.
[8] H.-W. Hung and A. Amiri Sani, "Brf: Fuzzing the ebpf runtime," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 1152–1171, 2024.
[9] "ebpf selftests," https://github.com/torvalds/linux/tree/master/tools/testing/selftests/bpf.
[10] V. Chipounov, V. Georgescu, C. Zamfir, and G. Candea, "Selective symbolic execution," in *Proceedings of the 5th Workshop on Hot Topics in System Dependability (HotDep)*, 2009.
[11] "Context confusion bug in bpf_common_lru_push_free," https://lore.kernel.org/bpf/CAPPBnEYgDtaOWYkk1rrMraZ4x3W-BQdi5nf2hURX8=xGxwr-1Q@mail.gmail.com/T/.
[12] "Context confusion bug in bpf_common_lru_pop_free," https://lore.kernel.org/bpf/CAPPBnEZBUWnRQ+mWpHXn2t7cfn=RVuWQZ_ojQte0gQpksus1Gw@mail.gmail.com/T/.
[13] "Context confusion bug in bpf_percpu_lru_push_free," https://www.spinics.net/lists/bpf/msg124825.html.
[14] "Context confusion bug in percpu_counter_add_batch," https://lore.kernel.org/bpf/CAPPBnEaJRP5YZTQ-xY4gt6eYxS8_WG0e4pwDYXvn5OtrjyFzaw@mail.gmail.com/T/.
[15] "Context confusion bug in percpu_counter_sum," https://lore.kernel.org/bpf/CAPPBnEZQOVp19ESv35wZpmXzcrJmcCdt-H_0KirAtA6SCdMqoA@mail.gmail.com/T/.
[16] "Context confusion bug in trie_update_elem and trie_delete_elem," https://lore.kernel.org/bpf/CAADnVQ+PM3htCksKHp0uN8k0Ke-v-j2K4xPPrxtuWZNyeDAZWA@mail.gmail.com/T/.
[17] "Nested locking bug in bpf_common_lru_pop_free," https://lore.kernel.org/bpf/CAPPBnEZQDVN6VqnQXvVqGoB+ukOtHGZ9b9U0OLJJYvRoSsMY_g@mail.gmail.com/T/.
[18] "Nested locking bug in bpf_common_lru_push_free," https://lore.kernel.org/bpf/CAPPBnEajj+DMfiR_WRWU5=6A7KKULdB5Rob_NJopFLWF+i9gCA@mail.gmail.com/T/.
[19] "Nested locking bug in bpf_percpu_lru_pop_free," https://lore.kernel.org/bpf/CAPPBnEaCB1rFAYU7Wf8UxqcqOWKmRPU1Nuzk3_oLk6qXR7LBOA@mail.gmail.com/T/.
[20] "Nested locking bug in pcpu_freelist_pop," https://lore.kernel.org/bpf/CAADnVQLAHwsa+2C6j9+UC6ScrDaN9Fjqv1WjB1pP9AzJLhKuLQ@mail.gmail.com/T/.
[21] "Nested locking bug in pcpu_freelist_push," https://lore.kernel.org/bpf/CAPPBnEYm+9zduStsZaDnq93q1jPLqO-PiKX9jy0MuL8LCXmCrQ@mail.gmail.com/T/.
[22] "ebpf program types documentation," https://docs.ebpf.io/linux/program-type/.
[23] "ebpf kprobe program type," https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_KPROBE/.
[24] "ebpf traffic control program type," https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_SCHED_CLS/.
[25] "ebpf perf event program type," https://docs.ebpf.io/linux/program-type/BPF_PROG_TYPE_PERF_EVENT/.
[26] "Attach points," https://ebpf.io/what-is-ebpf/#hook-overview.
[27] "ebpf helpers functions documentation," https://docs.ebpf.io/linux/helper-function/.
[28] "ebpf maps documentation," https://docs.ebpf.io/linux/concepts/maps/.
[29] "ebpf kfuncs documentation," https://docs.ebpf.io/linux/concepts/kfuncs/.
[30] "Unreliable guide to locking," https://www.kernel.org/doc/html/v4.13/kernel-hacking/locking.html.
[31] "Lock types and their rules," https://docs.kernel.org/locking/locktypes.html.
[32] "Locking lessons," https://github.com/torvalds/linux/blob/master/Documentation/locking/spinlocks.rst.
[33] K. Lu and H. Hu, "Where does it go? refining indirect-call targets with multi-layer type analysis," in *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, 2019, pp. 1867–1881.
[34] "Possible deadlock in bpf queue map," https://lore.kernel.org/bpf/CABcoxUbYwuZUL-xm1+5juO42nJMgpQX7cNyQELYz+g2XkZi9TQ@mail.gmail.com/.
[35] "Libbpf documentaton," https://libbpf.readthedocs.io/en/latest/.
[36] "ftrace documentaton," https://www.kernel.org/doc/html/v4.17/trace/ftrace.html.
[37] Syzbot, "Nested locking bug," https://syzkaller.appspot.com/bug?extid=a4ed4041b9bea8177ac3, 2024.
[38] "Lockdep documentaton," https://www.kernel.org/doc/Documentation/locking/lockdep-design.txt.
[39] G. Inc., "Syzkaller," https://github.com/google/syzkaller, 2016.
[40] Syzbot, "Bpf trace printk bug," https://syzkaller.appspot.com/bug?extid=c3740bc819eb55460ec3.
[41] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proceedings of the eighteenth ACM symposium on Operating systems principles*, 2001, pp. 73–88.
[42] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "Cp-miner: A tool for finding copy-paste and related bugs in operating system code." in *OSdi*, vol. 4, 2004, pp. 289–302.
[43] Z. Li and Y. Zhou, "Pr-miner: automatically extracting implicit programming rules and detecting violations in large software code," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5, pp. 306–315, 2005.
[44] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in linux: Ten years later," in *Proceedings of the sixteenth international conference on Architectural support for programming languages and operating systems*, 2011, pp. 305–318.
[45] T. Ball, E. Bounimova, B. Cook, V. Levin, J. Lichtenberg, C. McGarvey, B. Ondrusek, S. K. Rajamani, and A. Ustuner, "Thorough static analysis of device drivers," *ACM SIGOPS Operating Systems Review*, vol. 40, no. 4, pp. 73–85, 2006.
[46] A. Machiry, C. Spensky, J. Corina, N. Stephens, C. Kruegel, and G. Vigna, "{DR}.{CHECKER}: A soundy analysis for linux kernel drivers," in *26th USENIX Security Symposium (USENIX Security 17)*, 2017, pp. 1007–1024.
[47] J.-J. Bai, Y.-P. Wang, J. Lawall, and S.-M. Hu, "{DSAC}: Effective static analysis of {Sleep-in-Atomic-Context} bugs in kernel modules," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, 2018, pp. 587–600.
[48] A. Kadav, M. J. Renzelmann, and M. M. Swift, "Tolerating hardware device failures in software," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, 2009, pp. 59–72.
[49] J.-J. Bai, J. Lawall, Q.-L. Chen, and S.-M. Hu, "Effective static analysis of concurrency {Use-After-Free} bugs in linux device drivers," in *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, 2019, pp. 255–268.
[50] M. J. Renzelmann, A. Kadav, and M. M. Swift, "{SymDrive}: Testing drivers without devices," in *10th USENIX Symposium on Operating Systems Design and Implementation (OSDI 12)*, 2012, pp. 279–292.
[51] J. Liu, L. Yi, W. Chen, C. Song, Z. Qian, and Q. Yi, "{LinKRID}: Vetting imbalance reference counting in linux kernel with symbolic execution," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 125–142.
[52] D. A. Ramos and D. Engler, "{Under-Constrained} symbolic execution: Correctness checking for real code," in *24th USENIX Security Symposium (USENIX Security 15)*, 2015, pp. 49–64.
[53] D. Engler and K. Ashcraft, "Racerx: Effective, static detection of race conditions and deadlocks," *ACM SIGOPS operating systems review*, vol. 37, no. 5, pp. 237–252, 2003.
[54] J.-J. Bai, T. Li, and S.-M. Hu, "{DLOS}: Effective static detection of deadlocks in {OS} kernels," in *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022, pp. 367–382.
[55] 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*, 2019, pp. 1069–1084.

[56] H. Vishwanathan, M. Shachnai, S. Narayana, and S. Nagarakatte, "Verifying the verifier: ebpf range analysis verification," in *International Conference on Computer Aided Verification*. Springer, 2023, pp. 226–251.

[57] X. Wu, Y. Feng, T. Huang, X. Lu, S. Lin, L. Xie, S. Zhao, and Q. Cao, "{VEP}: A two-stage verification toolchain for full {eBPF} programmability," in *22nd USENIX Symposium on Networked Systems Design and Implementation (NSDI 25)*, 2025, pp. 277–299.

[58] L. Nelson, J. Van Geffen, E. Torlak, and X. Wang, "Specification and verification in the field: Applying formal methods to {BPF} just-in-time compilers in the linux kernel," in *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, 2020, pp. 41–61.

[59] J. Van Geffen, L. Nelson, I. Dillig, X. Wang, and E. Torlak, "Synthesizing jit compilers for in-kernel dsls," in *International Conference on Computer Aided Verification*. Springer, 2020, pp. 564–586.

[60] M. Böhme, V.-T. Pham, M.-D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, 2017, pp. 2329–2344.

[61] S. Österlund, K. Razavi, H. Bos, and C. Giuffrida, "Parmesan: Sanitizer-guided greybox fuzzing," in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 2289–2306.

[62] X. Zhu and M. Böhme, "Regression greybox fuzzing," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 2169–2182.

[63] M. Zalewski, "american fuzzy lop," https://github.com/google/AFL, 2021.

[64] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, "Afl++ combining incremental steps of fuzzing research," in *Proceedings of the 14th USENIX Conference on Offensive Technologies*, 2020, pp. 10–10.

[65] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 475–485.

[66] I. Yun, S. Lee, M. Xu, Y. Jang, and T. Kim, "{QSYM}: A practical concolic execution engine tailored for hybrid fuzzing," in *27th {USENIX} Security Symposium ({USENIX} Security 18)*, 2018, pp. 745–761.

[67] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbetta, Y. Shoshitaishvili, C. Kruegel, and G. Vigna, "Driller: Augmenting fuzzing through selective symbolic execution." in *NDSS*, vol. 16, 2016, pp. 1–16.

[68] P. Chen and H. Chen, "Angora: Efficient fuzzing by principled search," in *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2018, pp. 711–725.

[69] T. Yue, P. Wang, Y. Tang, E. Wang, B. Yu, K. Lu, and X. Zhou, "Ecofuzz: Adaptive energy-saving greybox fuzzing as a variant of the adversarial multi-armed bandit," in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 2307–2324.

[70] D. She, K. Pei, D. Epstein, J. Yang, B. Ray, and S. Jana, "Neuzz: Efficient fuzzing with neural program smoothing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 803–817.

[71] D. Wang, Z. Zhang, H. Zhang, Z. Qian, S. V. Krishnamurthy, and N. B. Abu-Ghazaleh, "Syzvegas: Beating kernel fuzzing odds with reinforcement learning." in *USENIX Security Symposium*, 2021, pp. 2741–2758.

[72] P. Godefroid, N. Klarlund, and K. Sen, "Dart: Directed automated random testing," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, 2005, pp. 213–223.

[73] D. R. Jeong, K. Kim, B. Shivakumar, B. Lee, and I. Shin, "Razzer: Finding kernel race bugs through fuzzing," in *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2019, pp. 754–768.

[74] M. Xu, S. Kashyap, H. Zhao, and T. Kim, "Krace: Data race fuzzing for kernel file systems," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1643–1660.

[75] P. Fonseca, R. Rodrigues, and B. B. Brandenburg, "{SKI}: Exposing kernel concurrency bugs through systematic schedule exploration," in *11th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 14)*, 2014, pp. 415–431.

[76] X. Zou, G. Li, W. Chen, H. Zhang, and Z. Qian, "{SyzScope}: Revealing {High-Risk} security impacts of {Fuzzer-Exposed} bugs in linux kernel," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 3201–3217.

[77] W. Chen, X. Zou, G. Li, and Z. Qian, "Koobe: Towards facilitating exploit generation of kernel out-of-bounds write vulnerabilities," in *Proceedings of the 29th USENIX Conference on Security Symposium*, 2020, pp. 1093–1110.

[78] 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*, 2023, pp. 87–92.

[79] 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, pp. 615–628.

[80] C. Peng, M. Jiang, L. Wu, and Y. Zhou, "Toss a fault to bpfchecker: Revealing implementation flaws for ebpf runtimes with differential fuzzing," in *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, 2024, pp. 3928–3942.

[81] D. Jin, A. J. Gaidis, and V. P. Kemerlis, "Beebox: Hardening bpf against transient execution attacks," in *USENIX Security Symposium (SEC)*, 2024.

[82] L. Gerhorst, H. Herzog, P. Wägemann, M. Ott, R. Kapitza, and T. Hönig, "Mitigating spectre-pht using speculation barriers in linux bpf," *arXiv preprint arXiv:2405.00078*, 2024.

[83] S. Wiebing and C. Giuffrida, "Training solo: On the limitations of domain isolation against spectre-v2 attacks," in *2025 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 2025, pp. 3599–3616.

[84] S. Chintamaneni, S. R. Somaraju, and D. Williams, "Unsafe kernel extension composition via bpf program nesting," in *Proceedings of the ACM SIGCOMM 2024 Workshop on eBPF and Kernel Extensions*, 2024, pp. 65–67.

[85] "Patch to fix nested locking bugs," https://kernel.googlesource.com/pub/scm/linux/kernel/git/ardb/linux/+/c83508da5620ef89232cb614fb9e02dfdfef2b8f.

[86] K. K. Dwivedi, "Resilient queued spin lock," https://lkml.iu.edu/2501.0/05336.html, 2025.

[87] Y. Hao, G. Li, X. Zou, W. Chen, S. Zhu, Z. Qian, and A. A. Sani, "Syzdescribe: Principled, automated, static generation of syscall descriptions for kernel drivers," in *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2023, pp. 3262–3278.

[88] Y. Hao, J. Pu, X. Li, Z. Qian, and A. A. Sani, "Syzspec: Specification generation for linux kernel fuzzing via under-constrained symbolic execution," 2025.