BRF: Fuzzing the eBPF Runtime

HSIN-WEI HUNG, University of California, Irvine, USA
ARDALAN AMIRI SANI, University of California, Irvine, USA

The eBPF technology in the Linux kernel has been widely adopted for different applications, such as networking, tracing, and security, thanks to the programmability it provides. By allowing user-supplied eBPF programs to be executed directly in the kernel, it greatly increases the flexibility and efficiency of deploying customized logic. However, eBPF also introduces a new and wide attack surface: malicious eBPF programs may try to exploit the vulnerabilities in the eBPF subsystem in the kernel.

Fuzzing is a promising technique to find such vulnerabilities. Unfortunately, our experiments with the state-of-the-art kernel fuzzer, Syzkaller, show that it cannot effectively fuzz the *eBPF runtime*, those components that are in charge of executing an eBPF program, for two reasons. First, the eBPF verifier (which is tasked with verifying the safety of eBPF programs) rejects many fuzzing inputs because (1) they do not comply with its required semantics or (2) they miss some dependencies, i.e., other syscalls that need to be issued before the program is loaded. Second, Syzkaller fails to attach and trigger the execution of eBPF programs most of the times.

This paper introduces the BPF Runtime Fuzzer (BRF), a fuzzer that can satisfy the semantics and dependencies required by the verifier and the eBPF subsystem. Our experiments show, in 48-hour fuzzing sessions, BRF can successfully execute 8× more eBPF programs compared to Syzkaller (and 32× more programs compared to Buzzer, an eBPF fuzzer released recently from Google). Moreover, eBPF programs generated by BRF are much more expressive than Syzkaller's. As a result, BRF achieves 101% higher code coverage. Finally, BRF has so far managed to find 6 vulnerabilities (2 of them have been assigned CVE numbers) in the eBPF runtime, proving its effectiveness.

CCS Concepts: • **Security and privacy → Operating systems security**; • **Software and its engineering → Software testing and debugging**.

Additional Key Words and Phrases: Fuzzing, eBPF

ACM Reference Format:
Hsin-Wei Hung and Ardalan Amiri Sani. 2024. BRF: Fuzzing the eBPF Runtime. *Proc. ACM Softw. Eng.* 1, FSE, Article 52 (July 2024), 20 pages. https://doi.org/10.1145/3643778

1 INTRODUCTION

Extended Berkeley Packet Filter (eBPF) is a rapidly evolving technology in the Linux kernel that enables generic programmability in the kernel space. Originally, the Classic Berkeley Packet Filter (cBPF) was developed specifically for filtering network packets. It allowed user-supplied programs to be loaded and executed in the kernel space to inspect packets and decide whether to allow or reject them. A virtual machine in the kernel interpreted the simple cBPF bytecode of the program. In 2014, with new instructions and an enhanced virtual machine, eBPF was introduced [23]. As a result of the greater generic programmability and persistent data storage (i.e., maps) [22], it has quickly gained traction in different domains in the kernel. Not only is it widely adopted in different

Authors' Contact Information: Hsin-Wei Hung, University of California, Irvine, Irvine, USA, hsinweih@uci.edu; Ardalan Amiri Sani, University of California, Irvine, Irvine, USA, ardalan@uci.edu.

ACM 2994-970X/2024/7-ART52
https://doi.org/10.1145/3643778

places in the networking stack, it is also integrated with kernel tracing, debugging, and auditing frameworks.

While eBPF brings extensibility and performance to the Linux kernel, it also introduces risks as user-supplied programs run in the kernel. To ensure that an eBPF program, potentially supplied by attackers, can run safely in the kernel space, the eBPF verifier checks the program before loading it to make sure it will not execute indefinitely or access invalid memory. Obviously, the correctness of the verifier is critical to the safety of kernel. Therefore, many recent works [1, 8, 11, 12, 14–16, 19, 25, 27, 28] have proposed different ways to test or verify it.

However, we note that eBPF has a significant footprint in the kernel beyond the verifier, i.e., the eBPF runtime, which executes an eBPF program after it passes the verifier. The runtime components mainly consist of the execution environment (i.e., the interpreter or the just-in-time (JIT) compiler) and functionality that cannot be realized using only eBPF bytecode (e.g., helper functions and maps). Therefore, it is critical to find and fix the vulnerabilities in the eBPF runtime to prevent exploits.

Fuzzing is a promising technique to find such vulnerabilities. Unfortunately, our experiments with the state-of-the-art kernel fuzzer, Syzkaller, show that it cannot effectively fuzz the eBPF runtime for two reasons. First, the aforementioned eBPF verifier rejects many fuzzing inputs because (1) they do not comply with its required semantics or (2) they miss some dependencies, i.e., other syscalls that need to be issued before the program is loaded. Second, Syzkaller fails to attach and trigger the execution of eBPF programs most of the times. We briefly discuss these issues here.

Program semantics. Similar to other language processors, inputs to the eBPF subsystem (i.e., eBPF programs) pass through both syntax and semantic checks, and are then converted into low-level machine code (if JIT is enabled). More specifically, input eBPF bytecode first go through a series of checks in the verifier. Not only does the bytecode have to conform to the eBPF instruction set format (i.e., syntax), the control and data flow also need to conform to requirements imposed by the verifier to safeguard the kernel (semantics).

Program dependencies. In eBPF, a series of preparatory syscalls are often needed to be made in order to correctly load the bytecode into the kernel. The sequence and arguments of these syscalls may depend on the eBPF program. For instance, loading a program that uses maps requires making syscalls to create compatible eBPF maps in the kernel in the first place. The program also needs to be rewritten to refer to the external resources such as the created maps, which is referred as *relocation* in eBPF.

Program execution. Finally, loaded eBPF programs need additional syscalls to be executed. Since eBPF programs are triggered by kernel events, they first need to be correctly attached to the entry points. Then, corresponding events need to be created to trigger the execution of the programs.

To demonstrate Syzkaller's ineffectiveness in fuzzing the eBPF runtime, we perform an experiment. During 48-hour fuzzing sessions, only 19.5% of `BPF_PROG_LOAD` syscalls succeed in passing the verifier. And the programs that pass the verifier are simple. For example, the average number of instructions in a program is 4.86, showing that 50% of the successfully loaded programs effectively contain less than 4 instructions to fuzz the runtime (the last instruction must be a `BPF_EXIT`). Another eBPF fuzzer recently released by Google, Buzzer [20], also shows to be ineffective in fuzzing runtime components. Only 0.1% of eBPF programs generated by the fuzzer's pointer arithmetic fuzzing strategy pass the verifier.

In this work, we set out to tackle these challenges to effectively fuzz the eBPF runtime. Our solution is BRF[1], a fuzzer which is able to generate fuzzing inputs that, on the one hand, satisfy both the eBPF semantic constraints and eBPF program dependencies, and on the other hand, are

[1]We have open sourced BRF at https://github.com/trusslab/brf.

expressive enough to explore different execution paths within the runtime. Moreover, BRF attempts to attach and execute the successfully loaded programs.

BRF incorporates three novel solutions. First, BRF produces semantic-correct eBPF programs efficiently by using an *iterative error message-driven incorporation of semantic rules* as well as *source-code level program generation/mutation*. By generating/mutating eBPF programs at the source-code level and compiling them into actual fuzzing inputs, the compiler automatically takes care of the basic semantics of a program. For example, a branch instruction will not jump to invalid locations, or an instruction will not access invalid stack memory unless we explicitly perform pointer arithmetics. To further comply with additional semantics imposed by the eBPF verifier, we use the error messages of the verifier in an iterative process to extract the rules and integrate them with the program generation/mutation logic. Second, to tackle the challenge of syscall dependencies to eBPF programs, we study the relationship between the preparatory syscalls and eBPF programs, and then, in addition to random syscalls, we also generate preparatory syscalls with constrained arguments. Finally, we generate syscalls to attach and trigger the eBPF programs.

Using extensive experiments, we show that 97.6% of the fuzzing inputs generated by BRF succeed in passing the verifier while only 19.5% and 0.1% of the fuzzing inputs generated by Syzkaller and Buzzer pass the verifier, respectively. Moreover, in BRF, a large percentage of these programs are successfully attached to corresponding entry points and subsequently executed. Overall, in 48-hour fuzzing sessions, BRF manages to execute 8× and 32× more eBPF programs than Syzkaller and Buzzer, respectively. Furthermore, the programs successfully loaded by BRF are more expressive compared to programs successfully loaded by Syzkaller, i.e., they include 3.4× more instructions, 27.4× more calls to helper functions, and 17.1× more use of maps. The programs are also more expressive compared to Buzzer as the fuzzing strategy only uses a single type of helper function and map with fixed argument. As a result, BRF can cover 101% more basic blocks in the eBPF runtime when compared with Syzkaller. Finally, BRF has so far managed to find 6 new vulnerabilities (2 of which are assigned CVE numbers), proving its capability in finding vulnerabilities in the heavily-shielded runtime components.

We make the following contributions in this work. (1) We show that existing fuzzers are inefficient in fuzzing the eBPF runtime components due to weaknesses in three major aspects: 1) generating semantic-correct eBPF programs 2) meeting the syscall dependency to load programs 3) attaching and executing the loaded programs. (2) We generate semantic-correct eBPF programs using an iterative error message-driven incorporation of semantic rules (which uses automation as much as possible) as well as source-code level program generation/mutation. (3) We develop a dependency-aware input generation by collecting the dependency information with the help of automation whenever possible. (4) We provide an extensive quantitative and qualitative comparison of BRF and key related work and show that fuzzing inputs generated by BRF can better cover the eBPF runtime components.

2 BACKGROUND

2.1 Workflow of eBPF

We have witnessed a wide and rapid adoption of eBPF in areas such networking, tracing, and security [6]. For different use cases, there exist corresponding program types (e.g., `BPF_PROG_TYPE_SOCKET_FILTER` for filtering packets and `BPF_PROG_TYPE_LIRC_MODE2` for decoding infrared signals). The program types limit the resources they can access (e.g., `BPF_PROG_TYPE_LIRC_MODE2` should not be able to access network packets). We further break down the workflow into three phases: loading, attaching, and execution.

Loading. An eBPF program that a user wants to execute first needs to be loaded into the kernel. This involves loading BPF type format (BTF) information, creating eBPF maps, relocating the eBPF program and finally loading the program. First, since an eBPF program may point to some variables in the kernel, BTF information is needed to resolve the references. Next, eBPF maps used in the eBPF program need to be created by calling BPF syscalls. Then, an eBPF program with instructions referring to external resources (i.e., kernel variables, maps) needs to be rewritten to point to the actual resources, which is also known as relocation. Finally, the eBPF verifier checks the program for safety. After that, it will be compiled by the JIT compiler if enabled. The kernel will return a file descriptor of the loaded program on success.
Attaching. Once the eBPF program is loaded, the user can attach it using the aforementioned file descriptor to hooks in the kernel, which will be the entry points of the program. The hooks can be functions in the network stacks, kernel functions, security hook, etc.
Execution. Once the aforementioned hooks are triggered in the kernel, the attached program is executed either by an interpreter or natively if the in-kernel eBPF JIT compiler is enabled. Depending on the hook, different data will be passed to a program as the argument, which is called *context*. During execution, the eBPF program can store or read data in eBPF maps. It can also interact with the kernel through a predefined set of *helper functions* in the kernel, which allow a program to, for example, access maps, retrieve kernel information, or print messages. Finally, an eBPF program may return an integer value, which will be interpreted by the kernel according to the program type. For example, a socket filter program may return 0 to instruct the kernel to drop the packet.

2.2 eBPF Verifier

To ensure programs supplied by user space programs can be safely executed in the kernel, eBPF verifier statically checks them during loading. It mainly prevents unbounded execution, invalid jump, invalid memory access, and leak of sensitive kernel data. To do so, the verifier first checks if a program contains loops in the control flow. Then, it walks through the instructions and performs checks specific to the type of instruction. As it traverses the program, the verifier keeps track of the register state, which includes the potential ranges and types of values in the registers. Some examples of the value types are normal scalar values (`SCALAR_VALUE`), pointers to program context (`PTR_TO_CTX`), pointers to map (`CONST_PTR_TO_MAP`), and pointers to stack memory (`PTR_TO_STACK`). As a result, the verifier is able to determine whether a pointer dereference is safe. For example, an instruction will only access the valid fields in the context with correct permission, or only pointers to valid stack memory can be dereferenced. In addition, since the verifier can track the propagation of pointer values, leaking them to the user space (e.g., directly or indirectly through helper functions and return values) can be prevented. We will discuss the verifier rules in more depth in §4.

2.3 eBPF Runtime

We define the eBPF runtime components as the parts of the eBPF subsystem that are executed once an eBPF program passes the safety checks of the verifier. We identify four major components: JIT compiler, interpreter, eBPF maps, and helper functions.
JIT compiler. In systems running on supported architectures (e.g., x86 and ARM), verified eBPF programs can be further compiled into native machine code by the JIT compiler built into the kernel to speed up the performance.
Interpreter. If JIT is disabled or not supported by the architecture, the eBPF interpreter will be in charge of execution of programs, i.e., decoding eBPF bytecodes on the fly and executing them.
eBPF maps. One significant improvement of eBPF over cBPF is persistent storage that preserves data even after programs terminate. There are 31 different types of eBPF maps provided in Linux

v6.1. In general, they are key-value stores but differ in the underlying data structure (e.g., hash table, array, or ring buffer) or the type of elements being stored, e.g., generic data type, socket, or control group (cgroup). In addition to eBPF programs, maps can also be accessed by user space programs using BPF syscalls.

Helper functions. eBPF programs can interact with the kernel through helper functions, which are a predefined set of functions hard-coded in the kernel. There are 213 helper functions in Linux v6.1. A major portion of the helpers are used for accessing or manipulating eBPF maps. For example, `bpf_map_lookup_elem`, `bpf_map_update_elem`, and `bpf_map_delete_elem` are used for retrieving, modifying, and deleting an element in a map. Some examples for other helper functions include printing debug messages, getting the `task_struct` of the current task, and redirecting a packet to another network device. Note that the availability of helper functions depends on the program type as their usages only make sense under certain scenarios and privileges. Also, unlike eBPF maps, helper functions can only be invoked by eBPF programs.

We design BRF to fuzz all these runtime components. We specifically think that having the ability to fuzz eBPF maps and helper functions is important since as the eBPF technology finds new applications in the kernel, the number of eBPF maps and helper functions will for sure continue to grow. Indeed, the number of helper functions went up from 90 in 2019 [2] to 213 in 2022 [4].

3 OVERVIEW

3.1 Goals

In this work, we aim to fuzz the eBPF runtime components. The runtime primarily consists of the JIT compiler, the interpreter, the eBPF maps, and helper functions. To achieve this, we have three goals.

Goal I: Generating semantic-correct eBPF programs. Since most of the runtime components in the kernel space are only accessible through eBPF programs (except maps) instead of BPF syscalls from user space programs, we need to generate eBPF programs and let them test these components. Because the eBPF verifier enforces additional semantics in addition to C semantics on eBPF programs for safety reasons and reject incorrect programs, the randomly generated eBPF programs need to be semantic-correct in order to pass the heavy scrutiny of the verifier.

Goal II: Generating fuzzing inputs that meet syscall dependencies. A fuzzing input generated by a fuzzer is a program consisting of an eBPF program and some syscalls. For the randomly generated eBPF programs, to pass the verifier to be loaded into the kernel, a series of syscalls need to be made. The sequence and arguments of syscalls depend on the eBPF program. Therefore, to effectively test the runtime, a fuzzing input should at least contain these syscalls that satisfy the eBPF program dependencies.

Goal III: Generating syscalls that attach and trigger eBPF programs. After eBPF programs pass the verifier, to execute them, they first need to be attached to the hooks. Then, events corresponding to the hooks need to be generated in order to trigger them. Thus, a fuzzing input should also include syscalls that can attach eBPF programs and trigger the execution.

3.2 Design

Fig 1 shows the high-level idea of BRF. We first extract the eBPF domain knowledge, which includes the eBPF semantics and the syscall dependencies, through manual study of the source code and scripts that automatically parse the source code. Then, during the fuzzing phase, by leveraging the extracted domain knowledge, BRF is able to generate semantic-correct eBPF programs. It then generates the fuzzing input, which includes not only the eBPF program, but also the required

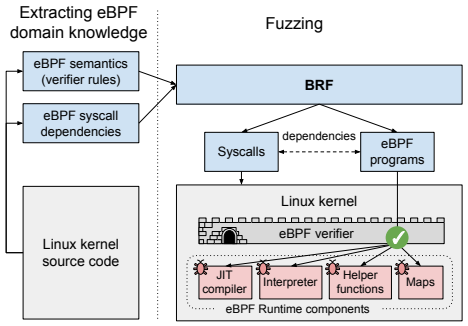

Fig. 1. The design of BRF

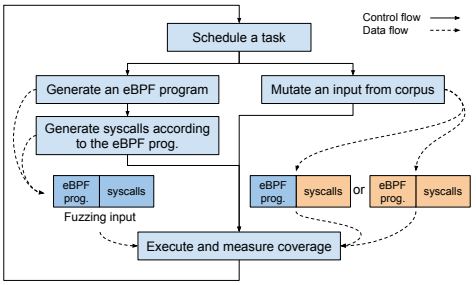

Fig. 2. The workflow of BRF. The orange blocks denote mutated inputs.

syscalls to correctly load, attach, and execute the eBPF programs. This way, BRF can reach deeply behind the eBPF verifier and effectively fuzz the BPF runtime.

3.3 Workflow

BRF is a coverage guided fuzzer as shown in Fig 2. In the main fuzzing loop, it generates a fuzzing input (which, as mentioned, consists of an eBPF program and a series of required syscalls.) This is done in two steps. First, BRF's scheduler generates a new eBPF program, or mutates an eBPF program in the corpus. Then, BRF generates different syscalls to form a fuzzing input. Next, we discuss these two steps in more detail.

In the first step, to generate a new semantic-correct eBPF program, BRF first randomly chooses a program type and a target helper function since helper functions are one of the key runtime components that we want to test. Then, it tries to generate the arguments of the helper function. For each argument, it generates the value based on the type. In general, there are three ways to generate a specific type of argument: First, some types of arguments can be generated by directly passing different values into the argument. Examples are scalar values or pointers to stack memory. Second, accessing different fields of the program context sometimes can result in different types of values. For instance, a socket filter eBPF program has `struct sk_buff` as the context. Accessing the field `sk` in this structure yields a `PTR_TO_SOCK_COMMON` value. Lastly, helper functions can return different types of values. Therefore, BRF may generate other helper function calls and their arguments recursively. Note that the generation logic is constrained based on the rules enforced by the verifier (§4) so that the resulting eBPF program will not contain incorrect semantics. The generated eBPF program source code will be compiled into eBPF bytecode and also serialized and stored into the corpus.

In the second step, BRF generates a user space program with different syscalls (i.e., fuzzing input). It generates syscalls necessary to load, attach and execute the eBPF program. It generates the arguments of these syscalls according to the eBPF program so that they are compatible (§6). It then generates and appends other random BPF syscalls to the fuzzing input as they can also access some runtime components.

After the eBPF program and the syscalls, serving as a fuzzing input, are generated, it will be executed. During the execution, BRF gathers coverage information to guide the fuzzing. That is, if a fuzzing input triggers new coverage, it adds the input to the corpus and mutates it to facilitate exploring different execution paths.

If the scheduler chooses to mutate a fuzzing input from the corpus, it may decide to mutate the eBPF program or the syscalls. To mutate an eBPF program, BRF first randomly selects an argument

Table 1. Types of error messages in the eBPF verifier.

Type of error msg.	Syntax	Semantic	Verifier error	Other
# error msg.	34	226	38	20

of a helper function call. Then, it randomly generates the argument again using the same logic for input generation. It then compiles the mutated source code to form the new mutated eBPF program. Since maps may be inserted or removed from the mutated eBPF program during this process, BRF needs to modify the syscalls associated with the eBPF program accordingly so that it can still load the eBPF program correctly. On the other hand, if the scheduler decides to only mutate the syscalls, only the randomly generated syscalls and their arguments can be mutated to ensure the eBPF program can still be loaded after the mutation.

4 GENERATING SEMANTIC-CORRECT eBPF PROGRAMS

To improve the effectiveness of the fuzzer in testing runtime components, generating semantic-correct eBPF programs is critical. Although eBPF programs can be written in C and compiled into executable using LLVM, the verifier imposes many additional semantics to ensure they can be executed in the kernel space safely. Not only are there many semantic checks, but they are also mostly *path sensitive*, making the verifier logic very complex. In the Linux kernel v5.15 source code, the main file that contains verifier logic has about 14,000 line of code, not to mention many of other program-specific or map-specific logic scattered elsewhere in the kernel.

Key approach 1: iterative error message-driven incorporation of semantic rules. Studying the verifier in order to identify all the semantic checks is a daunting task. Fortunately, we have two observations that allowed us to systematically tackle this challenge. More specifically, we noticed that whenever errors occur in the verifier, corresponding error messages are printed. As shown in Table 1, the error messages mainly include instructions with incorrect syntax, semantic violations, and verifier internal errors. Therefore, our first key approach is to use the error messages to enumerate all the semantic rules in the verifier.

We aim to only find the rules that prevent us from effectively fuzzing the runtime and then constrain our eBPF program generation logic according to these rules. We first annotate the verifier error messages with line numbers and start running the fuzzer without implementing constraints. That is, the fuzzer randomly generates helper functions and their arguments. Then, during the development of the fuzzer, we add a constraint associated with a verifier rule if the rules is triggered more than once per hour by the eBPF programs generated by the fuzzer. Note that, syntax rules are automatically satisfied by compiling the source code into eBPF bytecode.

By studying the 226 semantic rules, we are able to constrain the generation and mutation processes of the fuzzer. We explicitly incorporate constrains for 82 semantic rules into the fuzzer; most of the rest are implicitly taken care of by the compiler and the construct of the program we use. For example, the program will not contain unreachable code, unbounded loops, and the read-only frame pointer will not be written.

Key approach 2: source-code level program generation/mutation. We noticed that many semantic rules are not violated if the eBPF program is generated by a compiler from source code with normal construct. For instance, a normal program does not jump to an invalid address, indexing variable on the stack beyond valid range, or modifying the stack frame of another function without complex code that deliberately performs the operations. Therefore, our second key approach that further makes generating semantic-correct eBPF programs easier is source-code level program generation. Generating semantic-correct programs can be rather challenging for inputs generated

```
static const struct bpf_func_proto *
kprobe_prog_func_proto(enum bpf_func_id func_id
{
 switch (func_id) {
 case BPF_FUNC_perf_event_output:
    return &bpf_perf_event_output_proto;
 ...
 default:
    return bpf_tracing_func_proto(func_id, prog);
 }
}
```

Fig. 3. Simplified code showing how the verifier checks if a helper is available to kprobe eBPF programs.

```
static const struct bpf_reg_types
 scalar_types = { .types = { SCALAR_VALUE } };
...
static const struct bpf_reg_types *
 compatible_reg_types[__BPF_ARG_TYPE_MAX] = {
    [ARG_CONST_SIZE] = &scalar_types,
    ...
}
```

Fig. 4. Simplified code showing how the verifier checks if a helper is available to kprobe eBPF programs.

by a bytecode-level fuzzer. Take generating a random jump instruction as an example. A bytecode-level fuzzer not only needs to make sure the target is within the function, it also needs to keep track of the control flow of all basic blocks so that the jump does not introduce a loop or at the end leave some basic block never reached (i.e., dead code). In contrast, without using *goto* and *loop* in the source code, jump instructions are always valid.

We next discuss some important rules that we have dealt with. These examples show how we automate our solutions as much as possible. Although manual effort is inevitable in studying the rules and building the semantic awareness, we think it should be a one-time effort given that the semantics of eBPF should remain mostly the same.

4.1 Helper Function Availability

The helper functions available to each type of eBPF programs are different. This makes sense as different program types have different use cases and require different privileges. For example, for a socket filter eBPF program that requires almost no special privilege to be loaded, it should not be able to read arbitrary kernel memory using `bpf_probe_read_kernel`. Therefore, when the verifier encounters an eBPF call instruction to a helper function, it calls `get_func_proto`, a member function of the program type, to retrieve the function pointer to the helper function. An example is shown in Fig 3, which is the `get_func_proto` of kprobe eBPF programs. If the helper function is not available, the program will be rejected.

To deal with this, BRF first parses the source code of the Linux kernel and looks for the definition of `get_func_proto` for every program type. Sometimes, the `get_func_proto` of a program type may recursively call the `get_func_proto` of another program type. For example, `bpf_tracing_func_proto` shown in Fig 3 is the `get_func_proto` of the tracing type eBPF programs. Therefore, after BRF gathers all `get_func_proto` of every program type, those containing other `get_func_protos` are recursively substituted to produce the complete helper availability information. Then, during fuzzing, when BRF wants to insert a helper function, it only chooses randomly from the ones available to the program types.

4.2 Helper Function Arguments

eBPF verifier poses strict type checking on variables in the program. One such checks happens when passing them to helper functions. As opposed to the C semantics, where one can pass almost any type of variables to function arguments and implicit type conversion happens under the hood, variables passed to arguments of helper functions have to be of compatible types. For example, as Fig 4 shows, a constant size type argument only accepts scalar type variables. This suggests that a variable of any pointer type can never be passed to the argument because it not only makes no logical sense, but also creates a path to leak kernel pointers to user space.

To generate eBPF programs conforming to these semantic constraints, we first extract the compatibility information between variable types and argument types stored in the `compatible_reg_types`. Besides, BRF automatically parses the kernel source code to extract helper function prototypes, which include the type of arguments. Then, when generating an argument for a helper function, it randomly chooses a compatible type of variable according to the argument type declared in the prototype by directly creating one, calling another helper function, or accessing the program context.

4.3 Variable Safety Checks

To prevent unsafe memory accesses commonly seen in programs written in the C language, such as null pointer dereference or out-of-bound access, pointer type variables in eBPF need to be checked before being dereferenced. By comparing a pointer variable to a known value or range, the verifier (which is tracking the possible values of variables stored in the register state) is able to determine whether a pointer dereference is safe. Any unsafe pointer dereference causes the verifier to reject the eBPF program. The safety constraint not only applies to normal expressions but also to arguments of helper functions as pointers can be passed to them and then dereferenced in the kernel.

In detail, there are three different types of checks that are required to safely use different types of variables when passing them as helper function arguments.

Pointer. In the eBPF verifier, there are types of pointers that point to external resources, such as map values, sockets, memory locations or buffers. To make sure null pointer dereference will not happen in helper functions, these pointers need to be compared with `nullptr` before passing them as arguments.

Size. When a helper function takes a pointer to an external memory region as an argument, the argument following the pointer is the size, which is used in the helper function to access the memory. The size needs to be compared with a constant value that is smaller than the valid range of the memory to make sure out-of-bound memory access will not happen. Also, it might need to be compared with zero when the argument does not allow a zero-size access.

Packet. For a pointer to an external packet, it needs to be compared with pointers to the start and end of the packet to make sure the access is limited within the packet.

To satisfy these constraints, BRF keeps track of the valid values of pointers. Then, the safety checks are generated before using them as arguments in helper functions.

More specifically, when BRF tries to generate a helper function, it first generates its arguments, which could come from the return value of another helper function, the program context or direct allocation. Then, an `if` condition block wrapping the helper function call is generated. The predicates are filled with the safety checks of the variables passing to the arguments anded together, so that only when all arguments are safe to use, the helper function can be invoked.

For example, when a pointer to a map value is returned from a helper function, BRF knows that the valid size of the pointer should be the size of the value of the map. Therefore, when passing the pointer to another helper function that takes a pointer to a memory region argument and a size argument specifying the size of the memory that will be accessed in the helper function, two checks are added. First, BRF adds a check to make sure the pointer returned by another helper is not a `nullptr`. Second, the size argument is compared to the size of the map value, so that if another random value is passed to the size argument and is larger than the size of map value, the function will not be invoked.

Note that to prevent over-constraining (i.e., putting unnecessary checks on arguments), not all pointer arguments need to be checked. When tracking pointer values, BRF records whether a pointer can potentially be `nullptr` as some helper functions always return non-null pointers

according to the return value annotated in the prototype. Then, BRF generates a `nullptr` check only if a pointer argument does not allow `nullptr` and the pointer can potentially be a `nullptr`.

4.4 Program Context Access

eBPF programs are invoked with program contexts as arguments. Most of these are pointers to different structures depending of the type of the program and where they are attached. Not all members within the structure may be read or written, and the accesses to the contexts are checked by the verifier by calling `is_valid_access` to make sure they are not only within the structure, but also with the right permission.

We extract the permission and definition of contexts in different `is_valid_accesss`, and use this information to generate random but valid context accesses. When generating a variable for an argument by accessing contexts, BRF first determines if the argument is going to be read or written. This depends on if the argument is of `ARG_PTR_TO_UNINIT_MAP_VALUE` type, which suggests the pointer will be accessed in raw mode and could be written. Then, only the fields with the correct permission are used for random selection.

Note that unlike other variables that are declared and assigned right before the helper, there should be only one variable that accesses a specific field. Otherwise, the compiler optimization would introduce pointer arithmetic on the pointer to context, which violates another verifier rule. Therefore, variables generated by BRF using context accesses are inserted at the beginning of the program and then reused when needed.

4.5 Reference

eBPF programs have the ability to acquire references to some kernel resources through helper functions (e.g., acquiring a reference to a socket interface). Therefore, it is important that when a program terminates, references to resources are relinquished.

In BRF, in order to generate eBPF programs that satisfy the reference rules, a fix-up process is performed after an eBPF program is generated. More specifically, BRF goes through the helper functions in a program. If a reference acquired by a helper function never gets released by another helper function, a new reference-releasing helper function is generated and inserted into the program. On the other hand, when a reference-releasing helper function tries to release a variable that is not produced by a reference-acquiring helper function, BRF adds a helper function that returns a reference and then substitutes the original argument.

When developing BRF for this rule, we notice some false positives of the verifier. That is, an eBPF program conforming to the safety requirement is rejected due to the limitation of the verifier implementation. An example is shown in Fig 5. In line 2, `bpf_ringbuf_reserve` acquires a reference to an entry in a ring buffer, and therefore must be released before the program's exit. Although `bpf_ringbuf_submit` is called to release the reference in line 8 after checking if the reservation succeeds in line 7 (which is a necessary safety check), the verifier determines that there is reference leak if `v4` is null. Therefore, instead of releasing references before the program's exit, we generate reference releasing functions right after the use, which in this case is after line 4.

5 eBPF PROGRAM DEPENDENCIES

Being semantic-correct alone does not guarantee that an eBPF program will be successfully accepted by the verifier. It is also necessary for the program to have the correct preparatory syscalls to be called beforehand in order to load it into the kernel. These syscalls include BPF syscalls and other generic syscalls and depend on the eBPF program to be loaded. More specifically, in the loading process, these syscalls need to create compatible eBPF maps and relocate the eBPF program. Here we describe these syscalls and their dependencies to the eBPF program.

```
1   ...
2   v4 = bpf_ringbuf_reserve(&map_1, v2, v3);
3   if (v4)
4     v5 = bpf_ringbuf_query(&map_1, v4);
5   ...
6   if (v4)
7     bpf_ringbuf_submit(v4, 0);
8   return 1;
9 }
```

Fig. 5. An example semantic-correct eBPF program rejected by the verifier.

5.1 Creating Compatible Maps

Among the preparatory syscalls, `BPF_MAP_CREATE` syscalls need to be first called to create eBPF maps referred by eBPF programs. The arguments of the syscall decide the attributes (i.e., the type of the map, type of the key, type of the value, the maximum number of entries and the flag describing other properties) of the map to be created. Creating a compatible map can be done in two steps: (1) selecting a compatible type of map and (2) generating valid attributes for the map. The compatibility of the type of map depends on the type of the program and the helper function that takes the map as the argument. The attributes of a map are constrained by the type of the map in addition to the program type and helper functions. Failing to meet the constraints can result in the failure of map creation. In some cases, even if the map creation succeeds, the verifier later will reject the program during load time. We further describe the constraints (i.e., the dependencies between programs and maps) and how BRF satisfies them.

Helper function. During fuzzing input generation, an eBPF map is first generated when a helper function wants to use it as an argument. A helper function may only be compatible with a certain type of maps. An obvious example is that `bpf_ringbuf_output` only accepts `BPF_MAP_TYPE_RINGBUF` type of maps.

Thus, when selecting the type of map to be generated, BRF randomly chooses a compatible one using the compatibility information extracted from the verifier in the function `check_map_func_compatibility`.

Map type. For different types of maps, the attributes have different constraints and will be checked during the creation. For example, since a `BPF_MAP_TYPE_CGROUP_STORAGE` map only holds an entry local to a cgroup and is indexed by a 64-bit cgroup ID or `struct bpf_cgroup_storage_key`, the key size can only be 64 bits or the size of the key structure. Besides, since the number of entries is not configurable, the number of max entries in the arguments should be zero.

Therefore, to make sure `BPF_MAP_CREATE` syscalls succeed, BRF generates the attributes according to the constraints of different map types. There are four attributes; and the constraints we extract from the `map_alloc` and `map_alloc_check` functions of different map types can be generally described as followed: For the size of keys and the size of values, the constraints are the minimum size, maximum size and the alignment. For the flags, valid flags for a specific map are first separated into groups, where only one flag can be selected in each group. Then, these selected flags are `or`-ed together to form the final flag value. Finally, the constraint of the max entries is a single value that limits the upper bound of the random generated value.

Program type. The type of a program may also affect the types of maps that can be used (i.e., certain maps are not compatible with certain program types). For example, a tracing type program that will be attached to hooks that may sleep during execution can only use some basic hash and array maps. The type of program may also affect the flags of the map. For instance, the memory should be pre-allocated for perf event type eBPF programs, `BPF_PROG_TYPE_PERF_EVENT`. Therefore, the flag, `BPF_F_NO_PREALLOC`, shall not be used.

When generating a map, BRF first only selects from types of maps that can be used under the current program type, which is similar to how the compatibility of maps with helper functions is handled. Then, attributes are generated based on the constraints coming from the map type. For the attribute constraints introduced by the program types, BRF tries to fix the flags after all the attributes are generated depending on the programs type.

5.2 Relocating eBPF Programs

eBPF programs need to be relocated when they call other eBPF programs or there are instructions that refer to maps, external symbols, or global variables. In such a case, a user space program that wants to load the program first needs to create the maps or programs or resolve the reference. Then, the references in the instructions need to be updated.

BRF addresses the problem by inserting correct and immutable syscalls into the fuzzing inputs. To generate the correct syscalls, we leverage an eBPF utility library, `libbpf`. We directly invoke the API of `libbpf` that makes syscalls to load an eBPF program after parsing eBPF bytecode and other sections in the ELF file of the program. Note that these preparatory syscalls in the fuzzing inputs will not be mutated. As a result, instead of randomly generating these syscalls and hoping they are sufficient for loading an eBPF program, we can make sure an eBPF program will be loaded successfully in each fuzzing input.

6 EXECUTING eBPF PROGRAMS

To execute the eBPF programs generated by BRF, two steps need to be done by fuzzing inputs. eBPF programs supplied by user space programs will not be automatically executed after being successfully loaded into the kernel. They first need to be attached to hooks in the kernel. Then, when the events corresponding to the hooks happen, they will be executed.

6.1 Program Attachment

For some eBPF program types, the entry point information (i.e., where the program should hook to) is contained within the binary and therefore can be directly attached by calling `BPF_PROG_ATTACH` without specifying them in the arguments. A customized section in the ELF binary of a program is used to specify the type of the program. Sometimes, it may also contain the entry points. For instance, an eBPF program with an ELF section named `kprobe/sys_nanosleep` not only tells the user space programs loading it that it is a kprobe/eBPF program, it also says that the program should be attached to the entry point of the syscall, `nanosleep`. In this case, BRF uses a fixed hook for every program type when generating the binary.

However, for some types of programs, for flexibility reasons, the entry point information is not specified in the binaries. Instead, it is provided during attachment using the arguments of the syscall. An example is `BPF_PROG_TYPE_LIRC_MODE2`, which enables decoding infrared signal using eBPF programs. When attaching an LIRC eBPF program, an opened instance of the LIRC driver should be passed to the attach syscall to indicate from which devices the signal should be decoded by the program. Therefore, to properly attach these types of eBPF programs, BRF further generates syscalls required to create and open the resources, and then uses it for the attachment.

6.2 Triggering the Hooks

After the eBPF programs are attached, besides letting them get triggered opportunistically by random events in the kernel, BRF uses two methods to increase the probability of their execution.

First, BRF generates syscalls to trigger 7 types of eBPF programs (4 tracing-related, 2 network-related, and `LIRC`). For tracing type eBPF programs, we explicitly attach to events that happen frequently (e.g., tracepoints in the kernel scheduler, syscall entry points, or events signaled by

hardware periodically). For the two network-related programs, we call `recv` on a socket that we created and write data to the socket as well to trigger the execution. For `LIRC` programs, after attaching to the `LIRC` device, BRF generates syscalls that write signals to the device, so that the program will be invoked to decode the infrared signal.

Secondly, BRF leverages the BPF syscall `BPF_PROG_TEST_RUN` to deal with eBPF programs associated with subsystems that are hard to set up and test due to their complexity. Introduced for this exact same purpose, `BPF_PROG_TSET_RUN` facilitates testing of a subset of program types by building a simulated environment in the kernel and then test-running the eBPF programs. Therefore, BRF always generates a `BPF_PROG_TEST_RUN` syscall after loading and attaching programs.

7 IMPLEMENTATION

We implement BRF on top of Syzkaller in Go and C++ with 3000 LoC. We extracted the constraints from Linux v5.15 by manually studying the verifier and partially parsing the source code automatically. Finally, to collect coverage of eBPF programs for guiding the fuzzing, we extend the coverage collecting framework in the Linux kernel, `kcov`, and instrument the kernel.

7.1 Collecting Coverage of eBPF Programs

While Syzkaller already collects coverage information of the fuzzer process to guide the fuzzing, it does not include the coverage of eBPF programs. If not taken care of, eBPF programs that actually trigger new execution paths will not be prioritized, leading to sub-optimal fuzzing.

To collect coverage information of syscalls issued from the fuzzer process, Syzkaller leverages `kcov` in the Linux kernel. Through compiler instrumentation, a `kcov` function will be invoked for every edge in the kernel. For every thread that opens and enables `kcov`, the coverage information is then stored to a thread-specific memory region. In other words, only coverage of syscalls from user space processes are collected.

However, eBPF programs are triggered by different events depending on the program types and the hooks. Therefore, they normally execute in contexts different from the original fuzzer process that loads and attaches the eBPF programs, for example, kernel threads or interrupt contexts. As a result, even when a new helper function is invoked in an eBPF program, the eBPF program may not be prioritized since no positive feedback is generated.

Fortunately, Syzkaller supports collecting extra coverage information for other fuzzer-related processes by utilizing the *remote* coverage feature of `kcov`. This allows us to collect a kernel thread's coverage and then associate it with a user process. It requires manually instrumenting the source code, which involves three steps. First, the handle of the user space thread needs to be installed to the kernel thread of interest. We do so when a user space program loads an eBPF program into the kernel space. The handle is stored into `struct bpf_prog`. Second, in the kernel thread, we annotate when to start recording the coverage by calling `kcov_remote_start` with a handle. This is done at the beginning of the function `bpf_prog_run`, with the handle we store in the `struct bpf_prog`. Therefore, the coverage can be associated with the user space program regardless of where the eBPF program is executed. Finally, we call `kcov_remote_stop` to stop recording coverage when the execution of the thread leaves the area of interest, which is the end of `bpf_prog_run`.

We further extend the `kcov` remote coverage API to make tracing eBPF program coverage possible. Since the original remote coverage API allocates a large memory area for storing coverage in `kcov_remote_start` using `vmalloc`, it can only be invoked in context where sleep is permitted. However, this is not the case for most of the eBPF program entry points. Therefore, we create a preallocated version of the remote coverage API, which takes preallocated memory as argument when starting coverage collection. Then, for every eBPF program, we allocate a memory region during loading.

Table 2. Comparison of programs loaded, attached, and triggered by Syzkaller and BRF under 48-hour fuzzing sessions. The numbers are the average followed by the standard deviation. *# success keeps track of whether any eBPF programs referenced in a fuzzing input are executed. Therefore, it may contain duplicated eBPF programs. **Only the number of unique eBPF programs that are executed are counted, which is not available in Syzkaller.

	Program loading syscall			Program attaching syscall			Program execution	
	Total	# Success	Success rate	Total	# Success	Success rate	# Success*	Success rate**
Syzkaller	176k ± 14k	34k ± 7k	19.5% ± 3.8%	60k ± 17k	16k ± 5k	26.5% ± 0.7%	12k ± 1k	na
Buzzer	4,543k ± 289k	3k ± 0.2k	0.1% ± 0.0%	3k ± 0.3k	3k ± 0.3k	100.0% ± 0.0%	3k ± 0.3k	100.0% ± 0.0%
BRF	176k ± 9k	172k ± 10k	97.6% ± 0.8%	160k ± 8k	145k ± 8k	89.9% ± 1.2%	97k ± 8k	66.8% ± 3.7%

8 EVALUATION

8.1 Fuzzing Effectiveness

To evaluate the effectiveness of BRF in fuzzing the runtime components of the eBPF subsystem, we compare it with two open source eBPF fuzzers, Syzkaller and Buzzer [20]. Buzzer, developed by Google, aims to provide a generic fuzzing framework for users to explore different fuzzing strategies. While BPF fuzzer [1] is also open source, it has not been updated since 2019 and we can only build it successfully with Linux v4.2. Since eBPF has gone through substantial development in recent years, we would like to use Linux kernel v5.15 as a better fuzzing target. As a result, we decided not to include it in the comparison.

In the evaluation, we assign the same amount of resources to the three fuzzers, which are five virtual machines, each with eight fuzzer processes. Besides, since we are only interested in fuzzing the eBPF subsystem, only BPF related syscalls are enabled. To have a fair comparison, we enable all BPF-related syscalls for Syzkaller so that it is able to create the necessary resources for attaching programs and triggering the execution. For example, `socket` and `setsockopt` are enabled so that if a `BPF_PROG_TYPE_SOCKET_FILTER` eBPF program is generated, there are syscalls to create a socket and attach the program to the socket. For Buzzer, we enable the built-in fuzzing strategy that has successfully discovered a bug and generates more expressive inputs, *pointer arithmetic* [20]. For BRF, we only enable core eBPF syscalls as the fuzzer generates the necessary syscalls. In addition, since the JIT compiler is enabled by default on x86 for performance reasons, we report the fuzzing statistics under this configuration. However, when running fuzzing experiments to find vulnerabilities, we try both kernels with and without JIT compiler so that we can also test the interpreter. We run the fuzzer on a machine with Intel Xeon E5-2697 v4 CPU for 48 hours for five times and compare the results. Since all fuzzers evaluated are generator-based, we do not provide initial corpus. Besides, the corpus accumulated throughout a run is removed before starting the next to make sure each run is statistically independent.

Reaching the runtime components. First, we look at the three critical stages in the workflow of eBPF that will affect the fuzzing effectiveness: eBPF program loading, attaching and execution. Since many of the runtime components, such as the interpreter, JIT compiler, and helper functions, can only be accessed by eBPF programs, to achieve high fuzzing effectiveness, it is essential first for the programs to pass the verifier during the load time. As shown in Table 2, 97.6% of eBPF programs generated by BRF are able to pass the checks of the verifier, showing that these fuzzing inputs are both semantic-correct and meet syscall dependencies. Since we do not cover all semantic rules, there are still a small portion of fuzzing inputs that fail to pass the verifier. On the other hand, with only syntax-awareness and little knowledge of syscall dependencies, only 19.5% of programs generated by Syzkaller pass the verifier. For Buzzer, the pointer arithmetic fuzzing strategy generates random *alu* and *jump* instructions in the program. The fuzzing strategy has some degree of semantic awareness (e.g., the register values used need to be initialized). However, due to a flaw

in the implementation, all of the programs are rejected due to invalid shift immediate values. After we fixed the bug, 0.1% of programs generated are able to pass the verifier. Therefore, although it is able to generate 26× more programs than BRF, BRF loads 57× more programs in 48 hours.

Here we report five most violated verifier rules (>100 violations in 48 hour) that bottleneck the fuzzing of the runtime for Syzkaller: 1) calling into invalid destination 2) not having `jmp` or `exit` as the last instruction 3) calling into a `btf_id` which is not a kernel function 4) jumping out of range 5) calling a kernel function from non-GPL eBPF program. Note that these bottlenecks are relatively simple. The reason is that the semantic checks are layered and therefore the complex ones are still masked by the simple ones.

After eBPF programs pass the verifier, they will be compiled by the JIT compiler if enabled, and the JIT compiler will be fuzzed. Then, to exercise the interpreter (when JIT is disabled), helper functions and maps, programs first need to be attached. For Syzkaller, due to the fact that only a few eBPF programs are loaded successfully, the program attaching syscall has few valid program `fds` to start with. Since Syzkaller only generates program attaching syscalls by chance, only 26.5% of these syscalls succeed. Note that, these successfully attached programs of Syzkaller could be the same eBPF programs. Since Syzkaller generates syscalls randomly, it is possible that a fuzzing input retrieves an already loaded program pinned in BPF virtual file system. This also explains why the number of attached programs is larger than loaded programs for Syzkaller. While in BRF, we only count unique eBPF programs that are loaded. In contrast, since BRF generates program attaching syscalls for every fuzzing input, 89.9% of its attaching syscalls succeed. For Buzzer, it only generates one type of program and always uses the same function to attach and execute programs. Therefore, it is able to attach and execute programs successfully. Finally, BRF manages to trigger 66.8% of the attached programs. Whereas for Syzkaller, we are only able to collect the total number of eBPF programs executed with the possibility of duplication. All and all, the advantages of BRF in loading, attaching, and triggering eBPF programs result in BRF executing 8× more programs than Syzkaller and 32× more programs than Buzzer.

Table 3. Expressiveness of eBPF program generated and successfully loaded by Syzkaller and BRF, where M. denotes max.

	# Instructions		# Helpers		# Maps	
	Avg.	M.	Avg.	M.	Avg.	M.
Syzkaller	4.9 ± 0.1	16	0.4 ± 0.1	4	0.3 ± 0.0	4
Buzzer	493.6 ± 3.5	731	2.0 ± 0.0	2	1.0 ± 0.0	1
BRF	16.4 ± 0.1	314	11.0 ± 1.2	66	5.6 ± 0.7	18

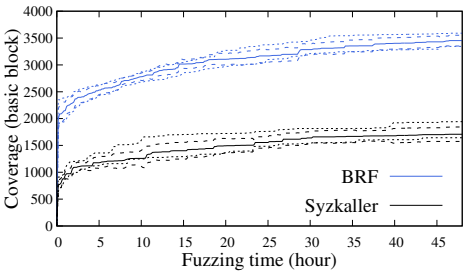

Fig. 6. Coverage of eBPF runtime components over time. Solid line: median; dashed lines: 68% confidence intervals; dotted lines: max/min.

Expressiveness of eBPF programs. Beside the success rate of loading, attaching, and executing eBPF programs, the expressiveness of the eBPF programs can also affect the fuzzing effectiveness. Therefore, we quantify the expressiveness of the generated eBPF programs by measuring the average number of eBPF instructions, average number of calls to helper functions, and the average number of usage of maps within successfully loaded eBPF programs. Our results, shown in Table 3, show that, on average, a program successfully loaded by BRF contains 3.4× more instructions, 27.4× more calls to helper functions, and 17.1× more use of maps, compared to a program successfully loaded by Syzkaller. For Buzzer, since it generated fixed instructions that log execution result, the programs generated always contain a map and two helper functions of the same type with

Table 4. Summary of vulnerabilities discovered by BRF.

	Verifier	Runtime components				New	Type of vulnerability
		JIT	Interpreter	Helpers	Maps		
CVE-2022-2905	✓	✓				✓	Out-of-bound access, info. leak
CVE-2023-0160				✓	✓	✓	Deadlock
Vulnerability 3				✓		✓	Deadlock, API misuse
Vulnerability 4				✓	✓	✓	Deadlock
Vulnerability 5				✓	✓	✓	Memory leak
Vulnerability 6				✓	✓	✓	Deadlock

fixed arguments. Therefore, although programs generated by Buzzer on average contains 493.6 instruction as opposed to 16.4 by BRF, the variable part only contains *jump* and *alu*. As a result, they cannot fuzz the eBPF maps and helper functions. The largest program generated by BRF can contain 66 helper functions and 18 maps. The higher effectiveness of BRF in both generating semantic-correct and expressive eBPF programs helps it reach broader and deeper in the eBPF runtime.

eBPF runtime coverage. We measure the ability of BRF and Syzkaller in covering different types of helper functions. eBPF programs generated by BRF use 155 different helper functions, while the ones generated by Syzkaller can only uses 134. Since the fuzzing strategy only focuses on the pointer arithmetic logic in the verifier, Buzzer uses only 1 type of helper function with fixed arguments. Therefore, this fuzzing strategy cannot fuzz eBPF maps and helper functions.

Code coverage. To see how the aforementioned key metrics in the fuzzing effectiveness translate into the ability of exploring more execution paths, we record the coverage of Syzkaller and BRF in the eBPF runtime components. (Since Buzzer only provides line coverage and has no coverage filter, we do not include its coverage into the comparison. But we think it is safe to assume that with only one type of map and helper function, it should have significantly less coverage in the runtime components.) We do so by only including the basic blocks in files that implement the eBPF runtime: JIT compiler, interpreter, helper functions and maps. Therefore, the BPF syscalls, verifier and glue code that connects eBPF subsystem with other subsystems are not included. The result is shown in Fig 6, where we find that during the 48-hour fuzzing sessions, BRF is able to continuously discover new paths. In contrast, the coverage of Syzkaller plateaus after 5 hours into the fuzzing. At the end of the session, BRF is able to explore 101% more basic blocks in the runtime components, showing its better ability in fuzzing the eBPF runtime.

8.2 Discovered Vulnerabilities

To find vulnerabilities, BRF relies on the same sanitizers in the kernel (e.g., KASAN, KMEMLEAK and lockdep) as Syzkaller. In our experiment, we do not encounter any false positive albeit some kernel sanitizers may produce. For false negatives, which happens when some kernel code is not covered, BRF tries to reduce it by improving the coverage. This is demonstrated in the experiment that shows BRF is able to catch vulnerabilities in the stable kernel that has been fuzzed by Syzkaller.

Table 4 lists the vulnerabilities found by BRF. We responsibly report vulnerabilities as we discover them. We reported the first two vulnerabilities and received CVEs. We reported vulnerability 3 to both the Linux kernel and vendor mailing lists. We went through a lengthy discussion and fixing process, which involved eBPF maintainers and even Linus Torvald. Eventually, the vendor did not assign a CVE after the fix was upstreamed. We have reported vulnerabilities 4 and 5 very recently. Therefore, we do not know if they will be assigned CVEs or not. We have not reported vulnerability 6 yet since we discovered a few days before this submission.

Table 5. Comparison of eBPF fuzzers. The program correctness is the percentage of programs generated that pass the verifier. *The fuzzer can generate syntax-incorrect programs after mutation. **The fuzzer has limited semantic awareness according to the fuzzing strategies implemented.

	BRF	Syzkaller [3]	Buzzer [20]	[1]	[15]	[19]	[11]	[16]
Open source	✓	✓	✓	✓				
Syntax-aware	✓	✓	✓	*	*	✓	✓	✓
Semantic-aware	✓		**			**		**
Program correctness	97.6%	19.5%	0.1%	na	na	0.77%	na	na
Fuzzing target by design	runtime	subsystem	verifier	verifier	subsystem	JIT	rBPF	verifier+JIT

9 DISCUSSIONS ON FUTURE MAINTENANCE OF BRF

In BRF, we build the semantic-correct program generation/mutation logic by studying the verifier as well as automatically extracting information (e.g., helper function definitions) needed by the logic. Therefore, as the eBPF subsystem evolves, a question we ask ourselves is how BRF will adapt to changes in the eBPF subsystem, and more specifically, whether the process will require hefty manual efforts. Fortunately, we think it requires moderate effort to keep it updated. Since we have already studied and incorporated many essential semantic rules, the fuzzer only needs update in two cases: (1) When there are new eBPF program types, helper functions and maps. (2) When there are new semantic rules enforced by the verifier. Updating the fuzzer for new programs, helpers and maps is straightforward and will need very little time/effort. Respecting newly introduced semantic rules requires more work as the new semantics needed to be added to the fuzzer. According to our measurement, the growth rate of eBPF semantic rules has been 15% year-over-year across 5 most recent LTS (long-term support) versions in 5 years. But note that not all rules need explicit support in the fuzzer as some may be satisfied automatically by the compiler. Moreover, as the eBPF technology matures, we think the growth in semantics rules will slow down.

10 RELATED WORK

Fuzzing the eBPF subsystem. There are a couple of solutions for fuzzing the eBPF subsystem [1, 11, 15, 16, 19]. BPF fuzzer [1] aims to test the eBPF verifier by leveraging LLVM fuzzer and sanitizer available in the user space. To do so, it compiles the verifier in the user space and lets the LLVM fuzzer perform mutation-based coverage-guided fuzzing. Overall, it manages to find one bug. Unfortunately, this approach cannot be applied if the fuzzing target it the runtime components as they are tightly integrated with the Linux kernel (e.g., the network stack and the tracing infrastructure) and recompiling them to run in the user space is impossible.

[15] is a syntax-aware fuzzer targeting the eBPF subsystem in the Linux kernel based on Angora [9], a mutation-based fuzzer that requires a set of initial inputs. Similar to [1], it also uses the sample eBPF programs in the Linux kernel source tree. It tracks interesting bytes that trigger new execution paths and then use gradient descent to guide the mutation. Based on the fact that it only found bugs in the `libbpf` in the user space, we think it has limited ability in generating semantic-correct fuzzing inputs. In addition, since the mutation logic is not syntax-aware nor semantic-aware, we expect the program to be less likely to be accepted by the verifier.

[19] is a bytecode-level semantic-aware fuzzer targeting the JIT compiler. It works by generating bytecode with some awareness about the semantics imposed on the register states. The generated eBPF program will first be checked by the verifier compiled in the user space using the approach in [1]. If the program is deemed valid by the verifier, it will be loaded into the kernel and JITed to see if it can trigger bugs. The experiment shows only 0.77% of generated eBPF programs are valid due to the limited semantics awareness, and the fuzzer manages to find one bug in the JIT compiler.

Another BPF fuzzer by Crump [11] is a syntax-aware differential fuzzer. By feeding the generated eBPF programs to two different execution environments (i.e., native execution of JITed code and executed by the interpreter) and comparing the results, BPF fuzzer can detect bugs in the runtime if the outputs are different. By using BPF fuzzer on RBPF, a user space BPF runtime implemented in Rust, it found two vulnerabilities. This approach requires the existence of two implementations (JIT and interpreter in this case) serving the same purpose (executing eBPF programs), which is not available for eBPF maps and code associated with different eBPF programs.

[16] is another eBPF fuzzer aiming to generate semantic-correct inputs on bytecode level. However, since it only focuses on ALU-related issues in the verifier, it does not generate helper calls or use maps, missing a significant portion of the runtime. As a result, the fuzzing only managed to trigger two previously discovered vulnerabilities in the verifier.

Compared to previous work, BRF is a semantic-aware and dependency-aware generator-based fuzzer that is able to generate eBPF programs that pass the verifier efficiently. Combining with efforts to attach and trigger the eBPF programs, BRF is able to reach deeply into the runtime and discover 6 new vulnerabilities. More important (and to the best of our knowledge), BRF is the first work that covers all major eBPF runtime components. Table 5 summarizes the comparison of BRF with these existing fuzzers.

Formal verification of the eBPF subsystem. In addition to fuzzing, formal verification is another approach used to improve the safety of eBPF subsystem by verifying the correctness of components in the eBPF subsystem or implementing the components with proven correctness [8, 12, 14, 27, 28].

Prevail [12] is an effort in implementing an alternative eBPF verifier in the user space with greater precision, which is adopted by eBPF for Windows [5]. JitK [28] implements an interpreter for cBPF with proven correctness. JitSynth [25] develops a tool that synthesizes eBPF bytecode into verified native RISC-V instructions. Jitterbug[14] models the eBPF JIT compiler in Rosette [24] and then uses it to verify the correctness of JIT compiler implementation of different architectures in the Linux kernel. [8] and [27] try to verify the range analysis mechanism (i.e., `tnum`) using formal verification methods. Interestingly, CVE-2022-2905 discovered by BRF is due to a flaw in `tnum`.

We believe that fuzzing the eBPF runtime is orthogonal to the verification of eBPF. Even in the future, when the verifier, JIT compiler and the interpreter can be implemented with proved correctness, we believe the rest of the runtime components are less likely to be verified due to their diverse functionality and increasing number.

Fuzzing language processors. Many efforts have been invested in improving fuzzing language processors [7, 9, 10, 13, 17, 18, 21, 26, 29], software that translate source code in high-level languages into lower-level languages. Examples are compilers, JIT runtimes, and interpreters. As mentioned earlier, eBPF is also a language processor.

These approaches can be categorized into mutation-based and generator-based. For mutation-based fuzzers, initial inputs are required to generate new fuzzing inputs, which often suggests that the fuzzers have limited knowledge about the syntax and semantics. For generator-based fuzzers, with some knowledge about the syntax or semantics, they are able to generate new inputs by themselves.

CSmith [29] is a C compiler fuzzer that performs differential testing. C code is randomly generated according to the grammar and then fed to different compilers. After compilation and execution, the results are compared to determine if there are bugs in the compilers. LangFuzz [13] is a mutation-based blackbox fuzzer that generates and mutates code fragments in the fragment pool, which is constructed by parsing codebases and test suites using a language-specific parser. IFuzzer [26] is a mutation-based fuzzer that try to generate new inputs using genetic programming. Angora [9] aims to improve the branch coverage by introducing several techniques that facilitate constraint solving without using symbolic execution. The techniques are context-sensitive branch coverage,

scalable byte-level taint tracking, search based on gradient descent, type and shape inference, and input length exploration. Nautilus [7] is a grammar-based fuzzer that does not rely on corpus. It combines coverage-guided feedback to achieve higher fuzzing performance. PolyGlot [10] improves the generic applicability of a semantic-aware fuzzer by performing mutation and analyses in IR level. It first generates an IR translator using the BNF grammar of the language. Then, inputs selected from corpus are lifted using the IR translator. Constraints mutation will produce syntax-correct inputs and a semantic validator will further fix semantic errors. Zest [17] facilitates fuzzing of the semantic-stage of language processors by preserving the validity of syntax of inputs and using the validity feedback as guidance in addition to code coverage. [18] is a JavaScript fuzzer that introduces aspect-preserving mutation that avoids destroying the semantic of corpus. By stochastically preserving structures and types in corpus, it has a better chance in generating syntax-correct and semantic-correct inputs. Gramatron [21] improves the efficiency of semantic-aware fuzzers by addressing two shortcomings of traditional parse tree-base fuzzer. It uses grammar automaton to avoid biased sampling and aggressive mutation to avoid small-scale mutation.

BRF adopts the generator-based approach with the guidance of code coverage. Besides, due to the complex eBPF semantics imposed by the verifier due to security reasons, we extract semantic rules from the verifier to make BRF semantic-aware, so that the fuzzing inputs are able to reach eBPF runtime efficiently.

11 CONCLUSIONS

This paper introduced the BPF Runtime Fuzzer (BRF), a fuzzer that can satisfy the semantics and dependencies required by the verifier and the eBPF subsystem. We addressed three important challenges in BRF: (1) generating semantic-correct eBPF programs, (2) generating syscall dependencies of eBPF programs, and (3) generate syscalls to attach and trigger eBPF programs. Our experiments showed, in 48-hour fuzzing sessions, BRF can successfully execute 8× more eBPF programs compared to Syzkaller and 32× compared to Buzzer. Moreover, eBPF programs generated by BRF are much more expressive than Syzkaller's and Buzzer's. As a result, BRF achieves 101% higher code coverage. Finally, BRF has so far managed to find 6 vulnerabilities (2 of them have been assigned CVE numbers) in the eBPF runtime, proving its effectiveness.

ACKNOWLEDGMENTS

The work was supported in part by NSF Awards #1763172 and #1846230 as well as Google's 2020 Android Security and PrIvacy REsearch (ASPIRE) Award.

REFERENCES

[1] 2019. BPF Fuzzer. https://github.com/iovisor/bpf-fuzzer.
[2] 2019. BPF-HELPERS - list of eBPF helper functions, Linux, 2019-03-06. https://web.archive.org/web/20190313070209/https://man7.org/linux/man-pages/man7/bpf-helpers.7.html.
[3] 2021. Syzkaller: coverage-guided Linux system call fuzzer. https://opensource.google.com/projects/syzkaller.
[4] 2022. BPF-HELPERS - list of eBPF helper functions, Linux v6.1, 2022-09-26. https://web.archive.org/web/20230715024726/https://man7.org/linux/man-pages/man7/bpf-helpers.7.html.
[5] 2022. eBPF for Windows. https://github.com/Microsoft/ebpf-for-windows.
[6] 2022. eBPF Summit. https://ebpf.io/summit-2022/.
[7] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. 2019. NAUTILUS: Fishing for Deep Bugs with Grammars.. In *NDSS*. https://doi.org/10.14722/ndss.2019.23412
[8] Sanjit Bhat and Hovav Shacham. 2022. Formal Verification of the Linux Kernel eBPF Verifier Range Analysis. https://sanjit-bhat.github.io/assets/pdf/ebpf-verifier-range-analysis22.pdf. (2022).
[9] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. In *2018 IEEE Symposium on Security and Privacy (SP)*. IEEE, 711–725. https://doi.org/10.1109/SP.2018.00046

[10] Yongheng Chen, Rui Zhong, Hong Hu, Hangfan Zhang, Yupeng Yang, Dinghao Wu, and Wenke Lee. 2021. One engine to fuzz'em all: Generic language processor testing with semantic validation. In *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 642–658. https://doi.org/10.1109/SP40001.2021.00071
[11] Addison Crump. 2022. Earn $200K by fuzzing for a weekend: Part 1. https://secret.club/2022/05/11/fuzzing-solana.html.
[12] Elazar Gershuni, Nadav Amit, Arie Gurfinkel, Nina Narodytska, Jorge A Navas, Noam Rinetzky, Leonid Ryzhyk, and Mooly Sagiv. 2019. Simple and precise static analysis of untrusted linux kernel extensions. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. https://doi.org/10.1145/3314221.3314590
[13] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *21st USENIX Security Symposium (USENIX Security 12)*. 445–458. https://dl.acm.org/doi/10.5555/2362793.2362831
[14] Luke Nelson, Jacob Van Geffen, Emina Torlak, and Xi Wang. 2020. Specification and verification in the field: Applying formal methods to BPF just-in-time compilers in the Linux kernel. In *Proceedings of the 14th USENIX Conference on Operating Systems Design and Implementation*. https://www.usenix.org/conference/osdi20/presentation/nelson
[15] Benjamin Curt Nilsen. 2020. Fuzzing the Berkeley Packet Filter.
[16] M. H. Noor, X. Wang, and B. Ravindran. 2023. Understanding the Security of Linux eBPF Subsystem. In *Proc. ACM Asia-Pacific Workshop on Systems (APSys)*. https://doi.org/10.1145/3609510.3609822
[17] Rohan Padhye, Caroline Lemieux, Koushik Sen, Mike Papadakis, and Yves Le Traon. 2019. Semantic fuzzing with zest. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 329–340. https://doi.org/10.1145/3293882.3330576
[18] Soyeon Park, Wen Xu, Insu Yun, Daehee Jang, and Taesoo Kim. 2020. Fuzzing javascript engines with aspect-preserving mutation. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1629–1642. https://doi.org/10.1109/SP40000.2020.00067
[19] Simon Scannell. 2020. eBPF Fuzzer. https://scannell.io/posts/ebpf-fuzzing.
[20] Juan José López Jaimez Simon Scannell, Valentina Palmiotti. 2023. Alice in Kernel Land: Lessons Learned From the eBPF Rabbit Hole. *Black Hat Asia* (2023).
[21] Prashast Srivastava and Mathias Payer. 2021. Gramatron: Effective grammar-aware fuzzing. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 244–256. https://doi.org/10.1145/3460319.3464814
[22] Alexei Starovoitov. 2014. BPF syscall, maps, verifier, samples, llvm. https://lwn.net/Articles/609433/.
[23] Alexei Starovoitov and Daniel Borkmann. 2014. net: filter: rework/optimize internal BPF interpreter's instruction set. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=bd4cf0ed331a275e9bf5a49e6d0fd55dffc551b8.
[24] Emina Torlak and Rastislav Bodik. 2013. Growing solver-aided languages with Rosette. In *Proceedings of the 2013 ACM international symposium on New ideas, new paradigms, and reflections on programming & software*. https://doi.org/10.1145/2509578.2509586
[25] Jacob Van Geffen, Luke Nelson, Isil Dillig, Xi Wang, and Emina Torlak. 2020. Synthesizing JIT Compilers for In-Kernel DSLs. In *Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II*. Springer, 564–586. https://doi.org/10.1007/978-3-030-53291-8_29
[26] Spandan Veggalam, Sanjay Rawat, Istvan Haller, and Herbert Bos. 2016. Ifuzzer: An evolutionary interpreter fuzzer using genetic programming. In *European Symposium on Research in Computer Security*. Springer, 581–601. https://doi.org/10.1007/978-3-319-45744-4_29
[27] Harishankar Vishwanathan, Matan Shachnai, Srinivas Narayana, and Santosh Nagarakatte. 2021. 'Semantics, verification, and efficient implementations for tristate numbers. *arXiv preprint arXiv:2105.05398* (2021).
[28] X. Wang, D. Lazar, N. Zeldovich, A. Chlipala, and Z. Tatlock. 2014. Jitk: a Trustworthy In-Kernel Interpreter Infrastructure. In *Proc. USENIX OSDI*. https://www.usenix.org/conference/osdi14/technical-sessions/presentation/wang_xi
[29] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294. https://doi.org/10.1145/1993316.1993532

Received 2023-09-28; accepted 2024-01-23