Summary

eBPF (extended Berkeley Packet Filter) is a programmable interface for the Linux kernel system. The eBPF technology facilitates developing new kernel functionality through eBPF programs while accounting for the security concerns that follow. Security considerations are addressed as part of the eBPF verification process. eBPF programs determined to be safe can then be loaded and executed in the Linux kernel, delivering newly developed kernel features, such as process tracing, efficient execution and more.

eBPF programs consist of assembly-like instructions that function through access to a stack, shared memory and helper functions providing easy access to kernel native function calls. The eBPF technology has been widely adopted and the newly developed functionality has seen widespread use in server infrastructure and network related software.

For this project we choose to study fuzzing, and how it can be applied to the eBPF domain. Fuzzing goes through six phases in order to uncover errors, vulnerabilities, or faults in the target. The first two phases revolve around identifying the target software, and the data format of the target software input. Phase three through five are run repeatedly in sequence. In the third phase, data is generated in the identified format. This is typically done by mutating existing well formatted inputs, or generating new ones based on a specification. The fourth phase is executing the target with the generated input. The target is executed with the generated input, and the behavior of the target is observed as the fifth phase. During this phase, interesting behavior is observed. Interesting behavior can be external program output, or internal behavior observed by inspecting the target source code. The information can be used to guide the fuzzer, or in order to uncover bugs. The sixth phase is inspecting the found bugs, and the impact they have on the target software.

This project covers the development of Buzzy, a blackbox fuzzer for eBPF technologies. Buzzy involves designing strategies for a targeted eBPF feature. These strategies are designed in order to test the robustness of the targeted feature. The approach is simple to extend, yet effective at uncovering underlying inconsistencies in the eBPF technology. Buzzy has found 3 bugs in the PREVAIL verifier, 2 of which have already been assessed and fixed. Buzzy has also found 2 bugs in uBPF, neither of which have been assessed or fixed.

In Section 1, we present motivation for our project and list our contributions. In Section 2, we give an overview of the eBPF technology. In Section 3, we present fuzzing theory. In Section 4, we design our fuzzing harness, covering and explaining our choices made during the project. In Section 5, we present the implementation of our fuzzing harness. In Section 6, we conduct a set of experiments, and eveluate our harness based on the results and found bugs. In Section 7, we present work related to the project. In Section 8, we conclude on the usefulness of our fuzzing harness. In Section 9, we present what we consider the next areas for further development.

Buzzy: An Unguided Smart-Strategy Generation-Based Blackbox Fuzzer for eBPF Technologies

Tobias B. S. Hansen and Mikkel T. Jensen
Group: cs-23-ds-10-05

Aalborg University, Aalborg, Denmark
{tbsh18,mtje18}@student.aau.dk

Abstract. eBPF is a groundbreaking technology in the Linux kernel. It facilitates programmers to load programs into the kernel that, after a verification step, can JIT compile and execute the eBPF program. eBPF is widely used in server infrastructure and network management tools, as its place in kernel space facilitates tracing and real time enforcement of policies. Therefore, the correctness of eBPF is crucial. In this project, we develop Buzzy, a novel blackbox fuzzer for eBPF technologies. Buzzy uses a *strategy* based approach, where strategies are developed to target certain features in the chosen eBPF technologies. Buzzy is tested on the user space eBPF technologies, the PREVAIL verifier and uBPF virtual machine, maintained as part of the eBPF-for-Windows system. Results show that strategies are useful for generating more valid programs and for targeting certain bugs. Buzzy has found 5 bugs between PREVAIL and uBPF.

Table of Contents

1 Introduction

The Linux kernel abstracts the hardware and provides an API through which applications can run and share resources [27]. The functionality of applications is facilitated by system calls to a wide set of subsystems of the kernel. Each subsystem of the Linux kernel can be configured to varying degrees according to the needs of a user.

The maintenance of the Linux kernel is distributed among its developers and changes to the kernel and its subsystems involve changing the source code. Before some change or new functionality can be added to new kernel versions, the community of developers have to determine both whether the functionality is required and whether the new code can be safely added. This process means new kernel additions can take years before they are actually implemented in a new kernel version.

An alternative option is loading kernel modules, where new functionality can be added and loaded without making changes to the kernel source code. Kernel modules have to be actively maintained such that new kernel versions do not break the module. A broken module or security errors could crash or corrupt the running Linux kernel as modules are loaded as part of the kernel.

eBPF (extended Berkeley Packet Filter) is a groundbreaking technology that enables a programmable interface where new kernel functionality can be added while also accounting for security considerations. eBPF programs can be created through various tools or programmed manually. When an eBPF program is passed to the Linux kernel, it is compiled to bytecode and verified. A JIT (Just-In-Time) compiler is used to compile the eBPF bytecode to native machine code, when the eBPF program is called through event hooks. This allows for efficient and secure code execution in kernel space.

eBPF is widely used in server infrastructure and network management software. Correctness of the eBPF technology is important such that programs passed to the kernel behave according to the eBPF specification. This is important as errors in eBPF program behavior could be exploited for malicious intent.

1.1 Contribution

In this project we develop a fuzzer for eBPF technologies. A fuzzer attempts to uncover vulnerabilities or similar in target software by repeatedly generating input, executing the target with the input, and observing for interesting behavior.

An overview of the technologies that facilitate our fuzzing approach can be seen in Figure 1. When the eBPF program bytecode has been successfully verified it can be JIT compiled into native machine code and executed. Error detection is implemented by checking the agreement of program validity between the target verifier and virtual machine.

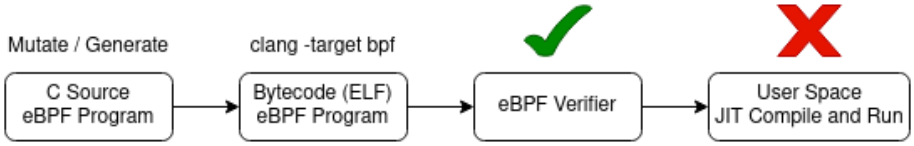


Fig. 1: An overview of the eBPF technologies underlying our fuzzing harness.

This project brings the following contribution to the eBPF community:

- **Buzzy**: An unguided smart-strategy generation-based blackbox fuzzer for eBPF technologies written in Rust. Buzzy is novel in its approach to fuzzing eBPF technologies as at is platform and error agnostic.
- **Complements prior work**: Buzzy complements prior eBPF projects. Buzzy uses the Rust crate, rBPF, that implements a high level interface for eBPF programming, in the generator component of Buzzy.
- **Future research**: Buzzy is a novel easy to use fuzzing harness. It allows future researchers to easily test user space eBPF technologies, a capability not previously seen.
- **Smart-strategy generation**: Buzzy implements an approach to eBPF program generation where a *strategy* can easily be defined. This allows programmers to easily implement a strategy targeting certain features. Feature targeting proved to be an effective way of uncovering bugs related to the given feature.
- **Found bugs**: As of writing, Buzzy has found three bugs in the PREVAIL eBPF verifier, and two bugs in the uBPF virtual machine for eBPF programs.

Acknowledgements

Thanks to our supervisors Danny Bøgsted Poulsen and René Rydhof Hansen, Associate Professors at Aalborg University, for their guidance through this project. Thanks to Lars Bo Park Frydenskov, Research Assistant at Aalborg University, for sparring in the early stages of the project. Thanks to Elazar Gershuni, the principal maintainer of PREVAIL, for quick answers to our eBPF specification questions. Thanks to Emil Fulei Lykke Aagreen and Frederik Arnfeldt Jensen, fellow students at AAU, for motivating us and making our studies fun.

Report Outline

In Section 2, we present the eBPF technology. We provide an overview of the technology, followed by different features implemented in the kernel to support it. We present the bytecode representation of the technology, and how it is compiled, verified, and run by the Linux kernel.

In Section 3, we present fuzzing theory. We present different areas of fuzzing, that should be considered for an effective fuzzing harness design. Different approaches to fuzz data generation are also presented.

In Section 4, we present the design of our fuzzing harness. This section follows the previously presented theory of fuzzing. We present our design choices, and reason why these choices were made.

In Section 5, we present implementation details of our fuzzing harness. The section starts with the information required such that our results can be replicated. This is followed by implementation details of the components we implement for our fuzzing harness.

In Section 6, we present the evaluation of our fuzzing harness. We first look into the ratio of valid and invalid programs generated by our harness. This is followed by a presentation of the bugs found by our fuzzing harness. The section ends with a discussion of our design choices in hindsight.

In Section 7, we present work related to the project. In Section 8, we conclude on how useful Buzzy is, and whether it adheres to the set goals. In Section 9, we present what we consider to be the next steps for Buzzy.

2 eBPF - Linux Kernel Interface

In this chapter we present the details of the eBPF technology. eBPF enables a programmable and flexible kernel interface such that users can make high performance kernel applications. Kernel level applications have access to privileged operations and information; to ensure the security of the kernel, eBPF programs are limited according to certain conditions. To keep the execution of eBPF programs safe, these conditions are verified by the in-kernel verifier. Programs verified as safe to run can then be run in kernel space by the in-kernel JIT compiler.

Chapter Outline

In Section 2.1, we present a general overview of the eBPF technology and its place in kernel space. In Section 2.2, we present different approaches to writing eBPF programs. In Section 2.3, we present features implemented in the Linux kernel, that can be utilized when creating eBPF programs. In Section 2.4, we present the instruction set used by eBPF; this includes instruction encoding, registers, and the eBPF stack. In Section 2.5, we present the different compilation and verification steps of an eBPF program. In Section 2.6, we present different toolchains utilizing the eBPF technology.

2.1 Overview of eBPF

eBPF is a Linux kernel system that facilitates running eBPF programs in a sandboxed environment [7]. An overview of this process is shown in Figure 2 [7]. eBPF programs can be used to extend the capabilities of the Linux kernel without making changes to the kernel source code or loading kernel modules. eBPF uses a general purpose instruction set, usually written in a subset of C, that is compiled into eBPF bytecode instructions. After going through a verification

process, the eBPF instructions are then mapped to native opcodes for the kernel through the eBPF in-kernel JIT compiler. This compilation process ensures optimal execution performance of eBPF programs in the kernel. The verified program can be attached to different hooks, allowing different events in the kernel to call the eBPF program. Linux OS distributions come in many different versions, and each version might utilize different versions or implementations of eBPF components.

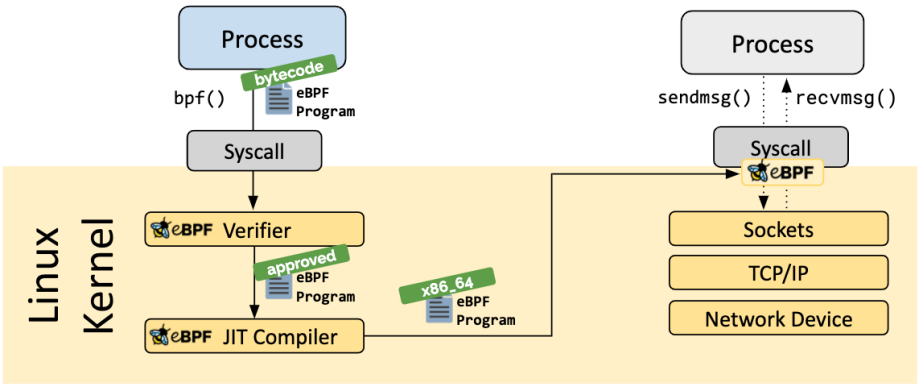


Fig. 2: Overview of the eBPF technology ecosystem [7]. The process on the left loads a program into kernel space. The eBPF program is verified by the kernel eBPF verifier. Another process can call and utilize the eBPF program, that is JIT-compiled to native machine code.

2.2 Writing eBPF Programs

There are a few high-level languages that support writing eBPF programs. The eBPF programs are then compiled to bytecode, accepted by the Linux kernel.

One way to write eBPF programs is by using the C library for eBPF programs. This allows programmers to use the constructs of C, such as if-statements and loops. The programs are then compiled to a common eBPF bytecode format that is accepted by the kernel as shown in Figure 3 [7].

BCC [13] is a Python library providing further abstraction when writing eBPF programs. BCC provides an environment, where eBPF programs can be written and debugged efficiently, as the abstractions provided by the library allows for easy loading, compiling, and running the eBPF program in the Linux kernel.

Other projects such as uBPF [14] and rBPF [24] facilitate writing eBPF programs in C and Rust, and running the code in user space virtual machines. The main idea behind these projects is to provide an easy way to test-run eBPF programs in user space.

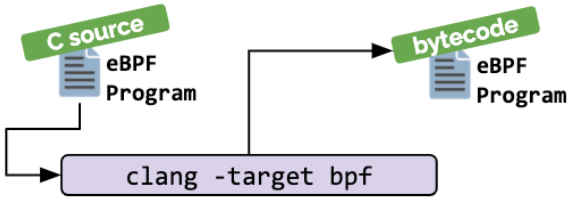


Fig. 3: eBPF program written in C, compiled to eBPF bytecode [7].

In the following sections we will provide a few examples of eBPF program snippets written in C and syscalls used for eBPF, to give an idea of how the technology can be used. The examples consist of, and is inspired by, examples used in the Learning eBPF book, by *Liz Rice* [27].

2.3 eBPF Structures

In the following sections we present different high level constructs that can be used in eBPF programs. This ranges from maps where data can be stored, to different helper functions giving access to kernel functionalities.

2.3.1 Program Context Arguments All eBPF programs have available a pointer to its context at register 1, referred to as the context argument. The structure pointed to by this register depends on the event that triggered the eBPF program or the program type. The program context could be a network packet, file descriptor, or map. The program type determines certain aspects of an eBPF program, such as possible hook attachments, available helper functions and the context at register 1.

2.3.2 Maps Maps are data structures used for sharing data between kernel and user space applications [27]. Maps can be shared across multiple CPUs or defined on a per-CPU basis. The data structures supported by maps include arrays, hash tables, ring buffers and stack traces. Maps are used to share information collected by eBPF programs or store various states. The information can be accessed by other eBPF programs, and also by programs written in user space.

As an example, a user space program might want to read configuration information retrieved by an eBPF application in kernel space. This is illustrated in Figure 4 [7], where hooks are attached to `sendmsg()` and `recvmsg()` syscalls, such that an eBPF program stores in- and outgoing network packet information for a process. The information can then later be accessed by the process on the left.

Maps are accessed by their *file descriptor*. Both eBPF programs and maps have an associated file descriptor, which is essentially in ID used for referencing. The kernel handles a reference count, which is used for file descriptors. When a

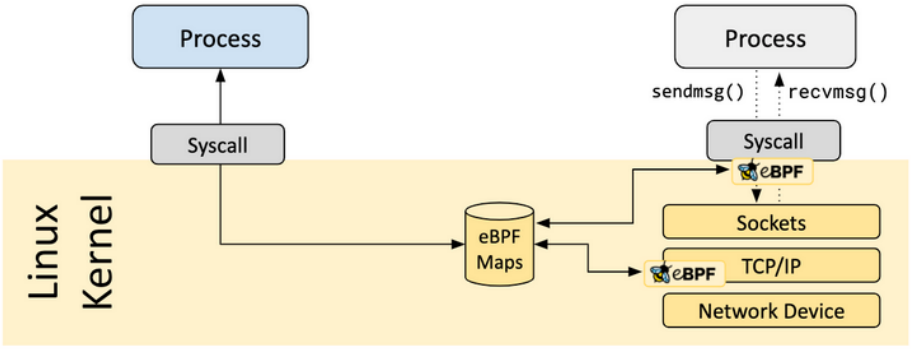


Fig. 4: Overview of two processes sharing information through an eBPF map. Hooks are attached to the syscalls `sendmsg()` and `recvmsg()`, storing information when a network packet is sent or received. The process on the left can then later access this information.

program or map is initiated in the kernel, the reference counter is incremented, and the value of the counter is returned as the file descriptor. When the program or map is closed, the reference counter is decremented. Individual programs can open and close maps, but maps are only destroyed when the global reference count reaches zero, because multiple eBPF programs can use the same map.

Example 1 (Initializing Maps).

A `bpf()` syscall can be made to load a map into kernel space from the terminal [27]:

```
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_ARRAY, key_size=4,
value_size=4, max_entries=4, map_name="array_map"}, 128)
```

This syscall is used to create a specified map type, in this case an array. The syscall returns a file descriptor for the map. The key size is set to four bytes. The value size of each entry in the array is set to four bytes as well. The map is set to have a maximum of four entries. This information is stored in a struct, making the information accessible for processes using the map.

2.3.3 Hooks eBPF programs are run when the kernel or an application passes a certain hook point or event [27]. The kernel has some predefined hooks, which include system calls, function entry/exit and events. Kernel probes (kprobes) are used for hooks in the kernel, allowing for eBPF programs to be attached to almost any instruction executed in the kernel code. Similarly, user probes (uprobes) are used to hook user applications. Tracepoints are marked locations in kernel code, that can also be used for eBPF hooks. They are similar to kprobes, but are stable across kernel versions. When an eBPF program has been attached to an event, the program will always run, no matter what triggered the event. It is also possible to create user defined hooks to attach eBPF programs almost anywhere.

Example 2 (Attaching eBPF Programs to Syscalls).

Attaching programs to hooks can be done in C [27]. In Listing 1 a function declaration is presented, attaching the defined function to the `execve` sycall. In this case, the code in the function is called whenever a program is executed.

```
SEC("ksyscall/execve")
int BPF_KPROBE_SYSCALL(kprobe_sys_execve){
    // Code ...
}
```

Listing 1: Defining a function, attached to the syscall `execve`.

2.3.4 eBPF Function Calls eBPF originally compiled functions into inline instructions, meaning that the compiled bytecode would not use jump instructions, to jump back and forth between function code [27]. As of Linux kernel version 4.16 and LLVM 6.0 eBPF now supports *BPF to BPF function calls* or *BPF subprograms*.

Tail Calls - eBPF programs can switch or pass execution context forward to another function, i.e. perform tail calls [27]. The motivation for tail calls is generally to avoid adding frames to the program stack. This is useful for eBPF as the stack is limited to 512 bytes.

Tail calls are made by calling the helper function `bpf_tail_call()`, where the current context is passed, along with an array of available functions to call, and an index for the array.

Helper Functions - eBPF programs can not use any arbitrary kernel function, as these are specific to the different kernel versions [27]. As part of a more stable API, the kernel makes available a set of helper function that the eBPF programs can use across any kernel. These helper functions include, a random number generator, timestamping functions and eBPF map access functions.

Example 3 (Function Calls).

Listing 2 is an eBPF program written in C, using the eBPF library [27]. The input of the function is a pointer to the program context. The helper function `bpf_printk` prints to the terminal. The function could be called as a tail call, in functions where the programmer want the callee program to print whenever the given event is triggered.

```
#include <linux/bpf.h>

int print_event_trigger(void *ctx) {
  bpf_printk("Event was triggered.");
  return 0;
}
```

Listing 2: An eBPF function used to trace when certain hooks are passed.

2.4 eBPF Instructions

Before eBPF programs are passed to the kernel, they are compiled into intermediate bytecode instructions. In the following section we present the low level components of the bytecode representation. eBPF programs consist of a program counter, a 512 byte stack, and eleven registers.

2.4.1 Registers eBPF programs are able to utilize ten general purpose registers, and a register for the stack pointer [35]. All registers are 64 bits wide. The registers are used as follows:

- `R0`: Return value of functions, and exit value for eBPF programs.
- `R1` to `R5`: Arguments for function calls.
- `R6` to `R9`: Callee saved registers, preserved across function calls.
- `R10`: Read-only frame pointer to the program stack.

An eBPF program can be executed with a packet, made accessible through its context register 1, which is initialized to point at the memory address at the start of the packet.

A part of the verification of an eBPF program is correct handling and usage of registers. As an example, register 0 must be initialized, i.e. if no value is written to the register, the program will be considered not valid. The eBPF verifier also tracks the usage of registers, such as correct initialization of registers before helper function calls.

2.4.2 eBPF Instruction Set eBPF supports two different instruction set encodings [35], basic instructions and a single wide instruction. The basic encoding, presented in Table 1, uses 64 bits to encode an instruction, written in big-endian. When written in little-endian the destination and source register are swapped, and the reading order of bytes is mirrored. A wide instruction of 128 bits exists for loading double words from immediate values. The wide instruction encoding adds 64 bits to the basic instruction encoding appended as an immediate value. The first 32 bits of the appended instruction must be zero, such that only the immediate value is used. A basic 64 bit instruction encoding consists of the following parts:

- `Opcode`: Operation to perform.
- `Destination register`: The destination register, ranging from 0 to 10.
- `Source register`: The source register, ranging from 0 to 10.
- `Offset`: Signed integer offset used for pointer arithmetic operations.
- `Immediate`: A signed integer immediate value.

eBPF operations are classified depending on the functionality of the operation. A list of operation classes are shown in Table 2. A full list of eBPF operations can be found at `kernel.org` [35].

8 bits	**4 bits**	**4 bits**	**16 bits**	**32 bits**
Opcode	Destination register	Source register	Offset	Immediate

Table 1: Bit encoding for a basic 64 bit eBPF instruction. For the wide instruction a 64 bit instruction is appended to the basic instruction encoding, where only the immediate value is used.

Memory operations are used to store or load values to the stack from a given register, or to load information from packets available through register 1. Memory operations can be performed using the sizes bytes, half words, words, and double words. These sizes perform operations using 1, 2, 4, or 8 bytes, respectively.

Jump operations are executed depending on some conditional between the destination and source register, e.g. is the value of the destination register equals to the value of the source register. If the conditional is evaluated as true, the program counter is incremented or decremented by the specified offset.

Arithmetic operations range from addition, subtraction, and multiplication to logic `or`, or `and` bit comparisons, as well as bit negations and bit shifts.

Class	**Opcode value**	**Description**
`BPF_LDX`	`0x01`	Load into register operation
`BPF_STX`	`0x03`	Store from register operation
`BPF_JMP`	`0x05`	64-bit jump operation
`BPF_ALU64`	`0x07`	64-bit arithmetic operation

Table 2: Selected classes of eBPF instructions, including loading and storing from register, jump operations, and arithmetic operations. The full list of classes can be found at `kernel.org` [35].

2.5 Compiling eBPF Programs

When an eBPF program has been written, it has to be compiled, verified and finally JIT-compiled into native machine code that the local machine can run [27]. This process is supported by the following steps:

1. Compile the source code into bytecode.
2. Load the program into the kernel.
3. Verify the compiled bytecode.
4. JIT-compile the bytecode to machine code.

The steps are presented in more detail in the following sections.

2.5.1 Compile eBPF Source Code Compiling eBPF programs is supported by *clang*. This can be done using the clang compiler from the LLVM project by passing the flag `-target bpf`. The result is an ELF (Executable and Linkable Format) object file.

Example 4 (Compiling eBPF Programs).

To give an insight into the bytecode produced when compiling eBPF programs, we present an example through Listing 3. This code is written in C, utilizing the C eBPF library. Two integers are initialized to 40 and 2, and the program returns an addition of these.

```c
#include <linux/bpf.h>

int func() {
  int a = 40;
  int b = 2;
  return a + b;
}
```

Listing 3: A small program initializing two integers to 40 and 2, then returning the addition of the two numbers.

Using clang to compile the program, the code in Listing 3 is compiled into the following bytecode:

```
b701 0000 2800 0000    -    mov64 r1, 0x28
631a fcff 0000 0000    -    stxw [r10-0x4], r1
b701 0000 0200 0000    -    mov64 r1, 0x2
631a f8ff 0000 0000    -    stxw [r10-0x8], r1
61a0 fcff 0000 0000    -    ldxw r0, [r10-0x4]
61a1 f8ff 0000 0000    -    ldxw r1, [r10-0x8]
0f10 0000 0000 0000    -    add64 r0, r1
9500 0000 0000 0000    -    exit
```

The above code moves the values 40 and 2 to register 1, each followed by a stack store operation. The two values are loaded from the stack into register `r0`, thereby initializing it, and `r1`. The value 2 from register `r1` is added to register `r0`, and the program exits.

ELF File Map Definition The maps data structure used in an eBPF program can be specified by the ELF file containing the eBPF program. Maps are defined by their attributes, `map_type`, `key_size`, `value_size` and `max_entries` [16].

The bytes for each attribute in the eBPF maps struct are specified in a `maps` ELF section. To utilize the maps data structure it requires a link between the ELF section containing the eBPF program and the ELF section containing the eBPF maps definition. Linking is done through ELF section relocations, which links a specific instruction in the eBPF program to the map that it accesses. When a map is recognized, it is available as the eBPF program context at register 1. The eBPF program expects register 2 to be the map key i.e. a pointer available for loading the map values to an offset on the stack.

2.5.2 Load Bytecode Into Kernel The compiled bytecode can be loaded into kernel to be run in the eBPF virtual machine. This can be done with different tools, or programmatically. Attributes are added to programs when they are loaded into the kernel such as the program name, type, and a file descriptor. The name and file descriptor can be used as identifiers for the program. The type indicates what kind of events the program can be attached to.

2.5.3 eBPF Bytecode Verifier A verifier is used to assure that the loaded eBPF bytecode is safe to run [7]. Among other properties, the verifier checks the following:

- The program does not harm or crash the system.
- No loops in the program, the program must terminate.
- The process loading the eBPF program must be a *privileged* process, unless *unprivileged* eBPF is enabled.

The verifier in the Linux kernel does this through two steps [36]. The first step consists of using a Control Flow Graph (CFG), determining whether the CFG is a Directed Acyclic Graph (DAG). This determines whether the program has unreachable instructions, and if the program loops indefinetely, both of which are disallowed. The verifier keeps track of the program state. At a certain instruction, if the program is in a state, that has previously been seen at this instruction, the branch is pruned reducing the searched state space.

The second step consists of traversing the CFG and descending every possible path to simulate the eBPF program. During the traversal, the verifier keeps track of all registers and the stack. The verifier has certain rules it applies, and checks whether the program adheres to the effects of these. A few of the rules are listed below:

- A register that has not been written to is not readable.
- When returning from a kernel function call, registers `R1` to `R5` are reset to unreadable.
- Registers are assigned certain types during verification, e.g. context pointers, `PTR_TO_CTX`, or scalar values, `SCALAR_VALUE`.
- Registers must have certain types when certain instructions are executed.

The verifier checks for more details, such as register liveness, data structure liveness, register parentage chains. The details can be found at `kernel.org` [36].

2.5.4 Bytecode JIT Compiling Before the local CPU can understand and execute the eBPF bytecode instructions it has to be translated to machine code that is native to the local CPU. eBPF bytecode is translated using JIT compilation. eBPF instructions were designed to be very similar to machine code, which means that JIT compilation is often a one to one translation. However, JIT compilation is able to provide some additional efficiency by optimizing across multiple instructions. This results in fewer machine code instructions and can result in a more efficient program.

2.6 Toolchains and Use Cases

eBPF programs extend the BPF technology, originally used for packet filtering, but is now used to program network, security, and observation applications for the Linux kernel. A wide range of toolchains provide services built using the technology. These toolchains are used by companies including Google, Facebook, Netflix, and Amazon. Below we present a few toolchains to give an overview of the application of the eBPF technology.

Kubernetes [10], or Google Kubernetes Engine (GKE), is a tool providing an environment for deploying, managing, and scaling of containerized applications. Kubernetes environments consist of multiple machines forming a cluster. Kubernetes provides a cluster management system, with benefits such as automatic management, monitoring, scaling, and rolling out updates. An example of Kubernetes in use is the audio-streaming service Spotify utilizing Kubernetes for its ability to roll out new services to hosts very quickly and efficiently.

Kubernetes utilizes both Cilium and Falco, and is heavily reliant on eBPF technology. An overview of this is shown in Figure 5, where a Kubernetes node utilizes Cilium. The container is compiled on the Linux OS node which provides the eBPF subsystem. One use case that eBPF facilitates is real time policy enforcement for filtering of packets.

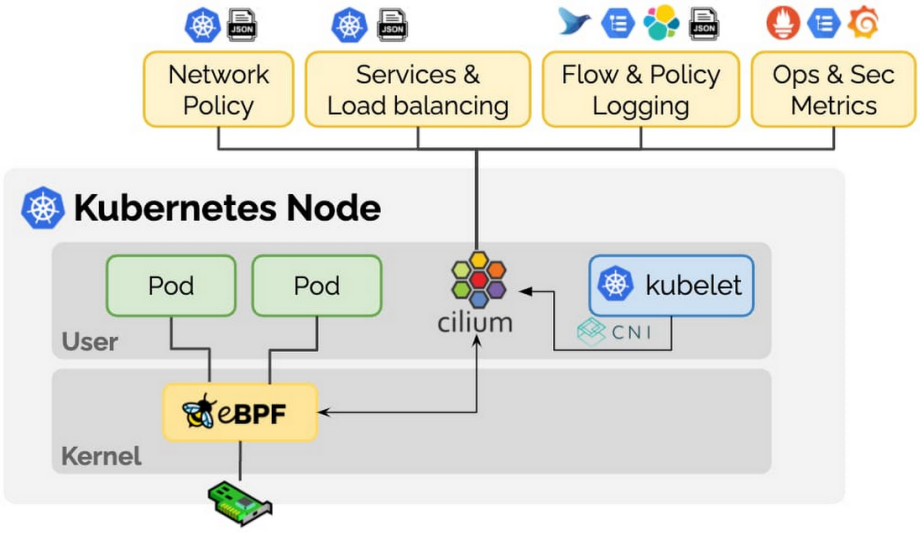


Fig. 5: A Kubernetes node utilizing Cilium [11]. The node is running on a Linux OS with the eBPF technology available. eBPF made tasks such as realtime policy enforcement possible, due to its placement in the kernel.

Cilium [33] is an eBPF-based dataplane providing network connectivity, observation and security. Cilium is built for scalable and highly dynamic cloud environments. Connections of environments can be observed, allowing for efficient visibility into applications and protocols. Cilium provides load balancing which is accelerated with the use of eBPF. One application of Cilium is Amazon Web services utilizing it for networking and security.

Falco [34] is a threat detection engine. The tool is used to detect threats at runtime by observing the behavior of applications. By observing system calls, Falco can assert the stream of calls against a powerful rules engine. When a rule is violated an alert is given. Falco checks for privilege escalation, creation of symlinks, ownership and mode changes, mutation of login binaries, etc. Alerts can be defined by end users, and typically range from logging using standard output to remote procedure calls to a client. Falco is used in applications such as GitLab, Shopify, and Google Cloud.

3 Fuzzing Theory

In this chapter we introduce fuzzing, a brute force vulnerability discovery method. Fuzzing is used to automatically find faults in software, by providing unexpected input and monitoring the software for interesting behavior, errors or crashes. This is done by repeatedly providing the target software with input data that has been mutated, or generated from a specification.

Chapter Outline In Section 3.1, we introduce fuzzing, the phases of fuzz testing, and design goals for effective fuzzing. In Section 3.2, we present three different approaches to fuzzing, namely black-, white-, and greybox fuzzing. These three approaches differ in the available information on the target software. In Section 3.3, we present different approaches to generating data. These strategies revolve around mutating existing target input, or generating new input based on a specification. In Section 3.4, we present differential fuzzing. An approach where external target behavior is compared against each other in an attempt to find interesting program behavior.

3.1 Introduction to Fuzzing

Fuzz testing, or fuzzing, is a process where inputs are repeatedly generated, executed by the target software, and observed, to find faults in the target [31]. Fuzz testing can be implemented on different levels, e.g. for a single function, a process, or a network protocol. The software that encapsulates generating input, running the target software, and observing outputs is referred to as a fuzzing harness. An overview of this process is shown in Figure 6.

Fuzzing can be compared to boundary value analysis, where the range of accepted values are identified, followed by the creation of tests where values

inside and outside the accepted range is tested [31]. These tests are designed to ensure that edge cases are handled properly, and exceptions are thrown where expected. Fuzzing is similar, but instead of focusing on boundary values, fuzzing attempts to generate a wide range of inputs to trigger undefined behavior.

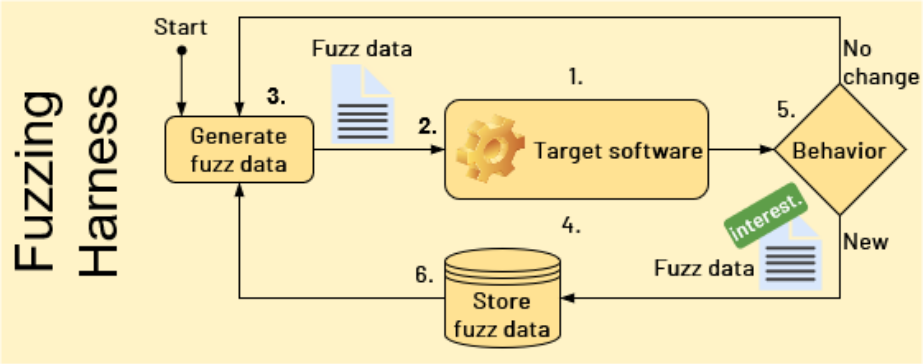


Fig. 6: Overview of a general fuzz test setup. Fuzz data is generated and passed to the target as an input. The target output is observed, and the fuzz data is saved if it causes new unseen behavior.

3.1.1 Fuzzing Phases Fuzz testing can be split into six phases [31], marked in Figure 6. Depending on the domain of the target, these phases can be implemented differently, possibly with a certain phase in focus. Network fuzzers have been designed to focus on learning the target input and state space automatically, in order to reduce the manual labor of this phase [26]. Fuzzers may also focus on efficient input generation, or executing targets fast and efficiently, without too much I/O overhead [8]. Monitoring and classifying faults can also be of focus, in order to handle errors that show up often, such that these do not bury other errors through taming [4].

1. **Identify target** relates both to the target software in its entirety, but also the entry point of the target itself. When identifying the target it can be necessary to identify whether the target should be fuzzed through a certain file format, a library used in the target, or a third option. This helps identifying the target input, and possibly a fuzzing tool to fuzz test the target.
2. **Identify inputs** aims to exploit the fact that software vulnerabilities often occur as a result of bad input sanitization or bad validation routines. These vulnerabilities are targeted by repeatedly providing the target with new input. Identifying a correct input format is important, as a bad input format can limit the fuzz test severely.

3. **Generate fuzzed data** depending on the chosen input target and format. This step is done through mutating existing predefined inputs, or generating inputs dynamically. No matter what approach is taken to generate the fuzzed data, this process should be automatic, as automization is a main principle of fuzzing.

4. **Execute target with fuzzed data** is strongly connected with the previous step, as these should be executed repeatedly one after the other. Depending on the target process, this step may involve opening a file, sending a packet, or running a process.

5. **Monitor for faults and exceptions** is crucial to a fuzz test setup. If a fuzzed data input causes a crash, it is important to be able to reproduce the crash, in order to determine what causes it. Monitoring can take many forms, depending on what faults are expected to be caught.
6. **Determine exploitability** is a phase considered after one or more faults have been found. This analysis is typically done manually by using debugging tools depending on the target. The extent of this step depends on the goals of the fuzz test. Fuzz testing in-house software might focus heavily on this step.

3.1.2 Effective Fuzzer Design When developing a fuzzer certain aspects should be taken into consideration to maximize the benefits and efficiency of the fuzzer [31]. Below we present six important areas of fuzzing harness design.

1. **Process States**: Being aware of the states in the target process can help increase the effectiveness of the fuzzer. As an example, knowing that a target compiler is split into a lexer, parser, and code generator can be used to target a certain stage of the compiler. If the lexer is the target, the input can be generated with a loose structure, possibly with keyword errors and similar. Assuming that the lexer and parser are well implemented, the target could be the code generator instead. A fuzzer for this state should generate more structurely sound programs, such that these get through the lexer and parser, in order to test the code generator.
2. **Code Coverage**: An efficient fuzzer should aim to maximize code coverage. Code coverage is a measure of what code has been executed during a run with a given input, e.g. executed lines or code blocks.
3. **Error Detection**: Different errors and faults show up in different ways. A program crash, and a thrown exception is not caught in the same way. To catch errors correctly, the expected errors should be designated, such that the fuzzer can be designed to catch these.
4. **Reproducibility and Documentation**: When an error occurs it is crucial to log the error such that the error can be reproduced. When running the target program on the logged error, the target should have the exact same behavior, as when the error was

first encountered. This assures that the exploitability of the error can be assessed, and the error can be documented.

5. **Reusability**: When designing a fuzzer, it is desired that the fuzzer can be applied to similar technologies, with very little implementation overhead. As an example, a fuzzer designed for a pdf-reader should be extensible to other pdf-readers. Keeping this in mind allows for greater code reusability.
6. **Resource Constraints**: Depending on the design of the fuzzer a lot of time might be needed. Implementing techniques for whitebox fuzzing is time consuming, and running the fuzzer on large programs also requires a lot of time for the fuzzer to solve constraints.

3.1.3 Fuzzing Limitations Fuzzing is used to find faults, exceptions, crashes or similar, but efforts are required to uncover the reason behind these [31]. Whether the error is caused by poor design logic or corrupt memory can not be decided by the fuzzer but requires manual inspection. In general, a fuzzer is not aware of the logic it triggers; so passing through an admin access point, while it should not be able to, will not be noticed by the fuzzer. These limitations should be considered when developing a fuzzer.

3.2 Fuzzing Approaches

In this section we present three different approaches to fuzzing: black-, white-, and greybox fuzzing. These approaches differ in the level of available information of the target software. A blackbox fuzzer can observe external behavior of the target program. Whitebox fuzzing utilizes target source code. Greybox is inbetween; the source code is not utilized, but information such as branch coverage is available, i.e. which code branches are hit during program execution.

3.2.1 Blackbox Fuzzing Blackbox testing refers to testing methods where only the external behavior of the target program is available [31]. This behavior can be the standard output of the program, memory usage, or network traffic.

A blackbox fuzzer either randomly mutates well-formed input, or generates inputs based on program specifications [3]. Using these specifications provides the blackbox fuzzer with otherwise limited target information.

The fuzzer may utilize external information of the target software to learn or determine how the generated input affected the target positively or negatively. If the fuzzer utilizes target output information it is regarded as a *guided* fuzzer. Blackbox fuzzers that do not utilize such information are *unguided*.

Example 5 (Blackbox Fuzzing Knowledge Limitations).

Blackbox fuzzing is limited by its level of knowledge. An example of this is shown in Listing 4 [3]. If a blackbox fuzzer is used to fuzz test the function `foo()`, the chance of hitting the `abort()` function is 1 in 2^{32}, assuming a 32-bit value of the integer `x`.

```
int foo(int x) { // x is an input
  int y = x + 3;
  if (y == 13) abort(); // error
  return 0;
}
```

Listing 4: A function assumed to be the target for a blackbox fuzzer. Finding the error caused by the `abort()` function has 1 in 2^{32} chance of happening, assuming `x` is a 32-bit integer [3].

Blackbox fuzzing is always applicable [31]. Since blackbox fuzzing does not require internal access or target knowledge, it is easy to apply to a target. A blackbox fuzzer can easily be adjusted to work with a different target similar to the original. Blackbox fuzzing does not have a metric of coverage, and it can be hard to determine if the target has been tested sufficiently.

3.2.2 Whitebox Fuzzing Whitebox testing refers to testing methods where the source code is available and utilized during testing [31]. This ranges from compile-time checkers, trying to identify vulnerabilities at compile time, to automated source code auditing tools, where the tool scans the source code in an attempt to find certain areas exposed to a certain vulnerability.

Whitebox fuzzing attempts to utilize methods from whitebox testing in order to trigger all code branches in the target program. Whitebox fuzzing may utilize symbolic execution and constraint solving [3], or taint analysis [15] with the key challenge of how to systematically explore the entire state-space of the target software.

Example 6 (Whitebox Fuzzing Search Space).

SAGE [9] is a whitebox fuzzer developed at Microsoft utilizing symbolic execution and constraint solving for whitebox fuzzing. The fuzzer symbolically executes the fuzz target using a predefined input. Every constraint, i.e. if-statement or while-loops, are picked up along the execution. These are then solved using constraint solving and the solution is applied to the input in order to trigger new code branches.

Consider the function `top()` presented in Listing 5 [9]. Assuming that the input starts out as `good`, SAGE will mutate the input such that every combination of the if-then-branches are hit, starting out by hitting zero branches. Mutating one character at the time, SAGE will eventually hit the `abort()` function, representing an error. The result is full coverage of the program state-space.

```
void top(char input[4]) {
  int cnt=0;
  if (input[0] == "b") cnt++;
  if (input[1] == "a") cnt++;
  if (input[2] == "d") cnt++;
  if (input[3] == "!") cnt++;
  if (cnt >= 3) abort(); // error
}
```

Listing 5: Assuming the the function `top()` is fuzz tested by a whitebox fuzzer, the goal is to cover every combination of if-then-branches. [9]

Whitebox fuzzing has the potential to result in complete code coverage, covering all possible execution paths [31]. The downside of whitebox fuzzing is the complexity of implementation, as the used techniques are complex and time consuming to implement. Another downside of whitebox fuzzing is that source code might not always be available, and the fuzzer is language specific.

3.2.3 Greybox Fuzzing Greybox testing is a step between black- and whitebox testing. It utilizes more than just the external information of the target software, yet the source code is not analyzed [31]. Greybox testing may utilize runtime information of the target binary, such as which blocks or statements are executed given an input.

Greybox fuzzers often utilize *code coverage* [42]. Code coverage is a measure of what code has been executed during a run of a program. Depending on the programming language, this information is accessible on different levels; code coverage can be tracked line by line, or code block by code block. Code coverage serves two purposes in fuzzers. It is a measure of how effective the fuzzer is, and the information can also be used to guide the fuzzer.

Example 7 (Greybox Fuzzing).

Consider Listing 5, and assume that the increments `cnt++` and the `abort()` function are on separate lines enclosed by curly brackets. Assume a greybox fuzzer is used to find the error caused by the `abort()` function, using line by line code coverage as a guide. If the input starts as `good`, the greybox fuzzer will count that each if-statement is hit, as well as the initialization of the `cnt` variable, but none of the increments are hit. As the fuzzer randomly mutates the input, it might generate the input `goo!`, now hitting one more line than before. Additional code coverage can be considered interesting behavior, and the input `goo!` is stored. Mutating one character at the time, and saving the results that hit new lines of code, the greybox fuzzer will eventually hit the `abort()` function. The input resulting in the error, will be logged for later analysis.

Greybox fuzzing is available for a wide range of targets, as the used techniques can be applied to binaries, and does not require the source code [31]. Greybox fuzzing is able to utilize target information through coverage metrics.

The downsides of greybox fuzzing is the complexity of extracting code coverage metrics, and using these efficiently to guide the fuzzer.

3.3 Generating Fuzzed Data

Methods for generating fuzzed data fall into two categories: mutation-based and generation-based. The approach to both methods can vary depending on whether the target accepts strings, files, or packets. Mutation-based fuzzing starts with a set of well-formed inputs and mutates these byte by byte in order to generate unexpected input for the target. Generation-based fuzzing builds the input bottom-up according to some specification. Some techniques fall outside these categories, such as generating completely random bytes and passing them to the target. The two approaches can also be combined, providing the structure from generation-based fuzzing, and field mutations from mutation-based fuzzing.

3.3.1 Protocol Awareness For both data generation methods, it is beneficial to design the fuzzer to be protocol aware [31]. A protocol is a convention or standard that enables communication or transfer of data between two computing end points. This can be how a computer reads from the hard drive, simple software functionality, or different server protocols for endpoint communication. The more that is known of the data structure, the better the input format specification can be determined.

If a fuzzer utilizes no knowledge of the target software input data structure it is referred to as a *dumb* fuzzer. Fuzzers utilizing knowledge of the input data structure are referred to as *smart* fuzzers. Common elements in protocol data specifications to be aware of are:

- **Name-value pairs**: Fields where certain values are accepted, e.g. `size=42`. Generating fuzz data for the value field is typically most beneficial.
- **Block identifiers**: Identifiers used to identify the type of data being represented in the data.
- **Block sizes**: Block sizes can be represented by a name-value pair. The name describes the incoming data type, and the value contains its length. The name-value pair is then proceeded by the data entry. Fuzzing can be applied to the length, making it larger or smaller than the actual length, to test the data handling of the receiver.
- **Checksums**: Checksums are used to check if data has become corrupt. It is vital to update checksums when these are present, to ensure proper processing of the input by the target.

3.3.2 Mutation-based Fuzzing Mutation-based fuzzing focuses on mutating a set of predefined well-formed inputs [31]. This set is often referred to as a *corpus*, and a single input from the corpus is a *seed*. The corpus often consists of seeds triggering different program behavior, i.e. they are well-formed and recognized by the target. By mutating these seeds byte by byte, the goal is to trigger new and unexpected program behavior.

AFL Mutations American Fuzzy Lop, or AFL, is a popular fuzzing tool. The mutation engine behind AFL has become an industry standard in fuzzing tools [18]. AFL applies a mutation algorithm utilizing the mutation methods listed below. The algorithms starts with small changes, i.e. bit flips, and moves towards more significant changes where multiple changes are applied at once.

- **Walking bit flips**: This technique sequentially moves over every bit in the input and flips it. One to four bits are flipped at a time.
- **Walking byte flips**: Similar to walking bit flips, flipping 8, 16, or 32 bytes in the input at a time.
- **Simple arithmetics**: To trigger more complex behavior, AFL adds or subtracts values to existing integers in the input file.
- **Known integers**: The last mutation is insertion of integers known to cause edge case behavior. These integers include -1, 256, 1024, MAX_INT, and $MAX\_INT - 1$.

3.3.3 Generation-based Fuzzing

In generation-based fuzzing the input of the target is studied in order to understand the input format [31]. An input specification, such as a grammar, can be constructed based on the input format. This grammar is then used to generate inputs bottom-up. In some cases, the grammar can easily be derived from existing specifications; a fuzzer designed to fuzz a compiler could use the grammar of the targeted language. Grammars are not the only available input specification to use. For simple input formats regular expressions or arbitrary structures could be used.

With a valid grammar, generated inputs are syntactically correct. Depending on the implementation of the grammar semantic constraints can also be applied. A function used to substitute non-terminals with terminals can be implemented to adhere to certain constraints, e.g. a certain range for integer values.

Using grammars facilitate the possiblity of using derivation trees. Derivation trees can be used to determine grammar coverage, i.e. what non-terminals have been expanded to certain terminals. With this information, the fuzzer can be guided to produce new and unique derivations. If a crash occurs and the input is saved, the derivation tree can be constructed, and possibly analyzed. The subtree resulting in the crash can be replicated and copied into other trees in an attempt to trigger new errors.

3.4 Differential Fuzzing - Case Study

In this section we use the fuzzing tool DifFuzz [25] as a case study to present the technique of *differential fuzzing*. DifFuzz is used as a side-channel analysis tool, to find possible side-channel vulnerabilities. Side-channel vulnerabilities are uncovered by observing non-functional characteristics of a program, such as time and memory usage. As an example, a password checker growing linear in time, with the amount of correct characters checked in the password, would be vulnerable to attack, as passwords can be guessed using a brute force algorithm.

The approach taken to uncover these vulnerabilities are presented in Figure 7. Typically differential fuzzing is used on two different versions of the same program, to observe differences in execution, which might be caused by bugs or implementation differences. DifFuzz observes the same program twice, but passes different *secret values* along with a shared public input, observing the difference in execution time and memory usage.

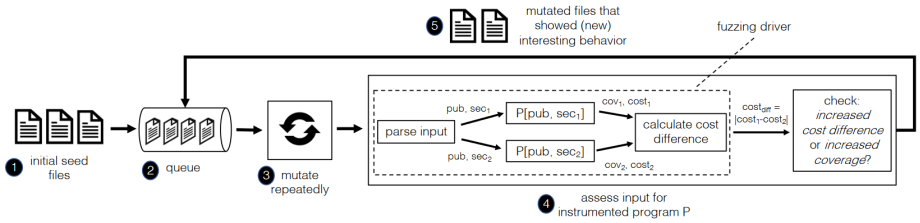


Fig. 7: Overview of the DifFuzz approach.

The fuzzing driver in Figure 7 is provided by the user of DifFuzz. The driver is responsible for parsing generated input into a shared public input and two secret values. The cost difference is measured by instrumenting the target programs, such that execution time and memory usage is recorded. The input is stored and considered interesting if the cost difference $cost_{diff} = |cost_1 - cost_2|$ is larger than previously observed. Stored inputs can then be manually inspected.

4 Fuzzing Harness Design

In the following chapter we present the design of our fuzzing harness for eBPF technologies. The chapter will focus on our approach to fuzzing, and the reason behind these choices.

As presented in Section 3.1.1, fuzzing is split into six different phases. Depending on the purpose of the fuzzer, different phases can be in focus when designing and implementing the fuzzer. The most important component of a fuzzer is the approach taken to generate fuzz data [18]. The approach covers both data generation algorithms and utilization of specifications derived from target knowledge. As our tool is a novel tool, we focus on the problems with regards to the fuzz data generation phase.

In Section 3.1.2, we presented six areas that affect the efficiency of a fuzzer. Each area should be accounted for when implementing a fuzzer. Improving any of these areas would result in improving the overall efficiency of the fuzzer. Considering the phase in focus, we aim to achieve improvements within the most relevant areas, such as code coverage.

This chapter reflects the six fuzzing phases. When presenting the choices made to design these six phases, we will focus on how we address the six areas of effective fuzzer design.

Chapter Outline

In Section 4.1, we present design goals of our fuzzing harness, and the arguments for our choices. In Section 4.2, we present an overview of our fuzzing harness approach, and the involved components. The rest of the chapter reflect the six fuzzing phases presented in Section 3.1.1. In Section 4.3, we present the chosen target software, and identify the input for the targets, ELF object files. In Section 4.4, we present our approach to generating the chosen input format. We present a generator component, able to generate eBPF instructions based on a chosen strategy. We also present a parsing component, able to produce an ELF file with the generated eBPF program. In Section 4.5, we present how we execute the targets, with the generated eBPF program. In Section 4.6, we present how we detect faults when executing the fuzz targets.

4.1 Design Goals

There are few fuzzers that have been developed for the eBPF technology. These are presented in Section 7. One is designed to target a specific type of out of bounds error for the in-kernel eBPF system [29]. Another is designed to target a user space eBPF implementation in Rust, used by the blockchain platform, Solana [1]. We aim to design a novel tool that is neither platform nor error specific.

The most impactful aspect of a fuzzer is the approach used for generating the fuzzed data [18]. We therefore focus especially on the fuzz data generation phase of our fuzzer design, as this is the area where there is most to be gained for a novel fuzzing tool. Below we briefly present an overview of how we address the six areas of effective fuzzer design for our fuzzing harness, and the reasoning behind our choices. Our design accounts for all six areas, but with a focus on the fuzz data generation phase. Therefore, we design our fuzzer especially to address the problem of code coverage within the eBPF domain, i.e. coverage of eBPF features. Additionally, we aim for a platform and error agnostic design, which greatly impacts the area of reusability. The following sections in this chapter will expand further upon these six areas.

1. **Process States**: We design our fuzzing harness to generate sufficiently complex structures to pass the basic parsing steps of our targets. It would otherwise be unlikely that random bytecode would form correct eBPF instructions. We do this as we aim to target the behaviors of eBPF program logic, which is determined after the eBPF program have been parsed and unmarshaled.
2. **Code Coverage**: We will take a smart generation-based approach to input generation. We will apply expert rules derived from the eBPF technology in an attempt to obtain feature coverage. Based on this domain knowledge we generate eBPF programs such that we are able to target the different features of the eBPF technology, e.g. maps, or the program stack.
3. **Error Detection**: We design our fuzzer such that it does not aim for any specific type of errors, catching arbitrary errors for later classification.

4. **Reproducibility and Documentation**: We design our fuzzer to store generated programs where the result is classified as an error. This will allow for replication of the error, by running the target software with the stored eBPF program. Erroneous eBPF programs can then be manually assessed by inspecting the target output.
5. **Reusability**: The fuzzer will be designed to be modular. Applied expert rules will not be dependent on the target, but rather the eBPF technology. This is done such that the developed fuzzing techniques are applicable to a wide variety of domain specific technologies built on top of eBPF.
6. **Resource Constraints**: We design our fuzzer with simplicity and ease of use in mind. Deploying our fuzzer on a new target eBPF technology should be straightforward for new users.

4.2 Fuzzing Harness Overview

In this section we will present an overview of our fuzzing harness and considerations relevant to the target process states. We will attempt to generate eBPF programs, such that they are parsed and unmarshaled correctly. This is done in an attempt to target the logical behavior resulting from eBPF program execution. The main idea of our fuzzing setup is being able to generate eBPF programs that are verified as safe to run, but produces an error when run on a virtual machine. To test this, we design a blackbox fuzzing harness. An overview of our fuzzing approach is presented in Figure 8.

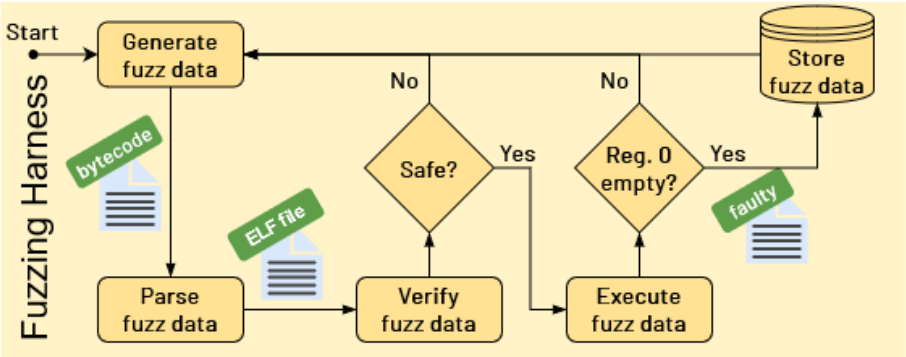


Fig. 8: An overview of our fuzzing harness setup. A generator generates an eBPF program, and it is parsed and written to an ELF file. This file can be read by a verifier. Depending on whether the eBPF program is safe to run, it is executed in a virtual machine. Programs causing faults when executed are stored for later inspection.

Square boxes with soft edges are components in the fuzzing harness setup. Diamonds resemble choices, and cylinders represent data storage. Our fuzzing harness implements or calls:

1. An eBPF program generator
2. An ELF file producer
3. An eBPF program verifier
4. An eBPF program virtual machine

We will design and implement the eBPF program generator, and produce an ELF file containing the generated program. We will target two eBPF technologies when deploying our fuzzer: an eBPF verifier to verify the generated eBPF program, and a virtual machine to execute eBPF programs verified as safe. The harness will be designed to be modular, such that the four components can be updated or possibly replaced entirely. This allows for testing of a wider range of eBPF technologies, and, as an example, targeting certain features in these technologies by writing a new generator component.

Arrows between components are also marked with the information flow between the components. The generator generates a struct, containing the generated eBPF program, parsed to an ELF file by the parser component. A verifier can then validate the generated bytecode, read from the ELF file. Valid programs can be executed and tested in a virtual machine. Depending on the virtual machine, various information about the program under execution can be stored. To test if the generated program behaves unexpectedly, we check to see if the virtual machine returns a value in register zero. No value will be returned if the program crashed or throws an exception during runtime. If register 0 contains no values, when the virtual machine has terminated, a fault exists in either the verifier or virtual machine, as they disagree on the correctness of the generated eBPF program. Generated eBPF program determined to be erroneous are stored for later inspection.

4.2.1 Other Harness Setups We use the harness structure as presented in Figure 8, but component modularity allows for other setups. We choose the given harness setup as it resembles how the program would be run in a real eBPF environment. Choosing a different setup could result in other sorts of conformance errors being found.

The harness could also be set up as a differential fuzzer to target multiple eBPF verifiers. The Linux verifier could be used as the oracle, meaning that it is assumed to produce the correct answers. The results of the other verifiers are then compared against the Linux verifier. If any result is different, the verifier is considered incorrect, and a bug or conformance error has been found.

4.3 Fuzz Targets and Inputs

In the following section we present our fuzz targets and the input chosen for the targets. Our approach to fuzzing will be a smart generation-based blackbox fuzzer, where we compare the results of a verifier against a virtual machine. The main idea behind the approach is that programs verified as safe by a verifier, should not be able to cause faults when run in a virtual machine, if the

two components agree on eBPF specifications. This can be tested by executing generated eBPF programs verified as safe in a virtual machine. If the virtual machine causes a fault, and does not return anything in register 0, i.e. the output of an eBPF program, an inconsistency must exist between the two.

For testing purposes it would be helpful to use the Linux verifier in user space such that it is isolated from the other eBPF kernel systems. Harnessing the Linux verifier in user space has been done by Trail of Bits [2] and Simon Scannell [29], by isolating the in-kernel verifier and its necessary dependencies. The steps required to isolate a Linux subsystem are extensive, requiring replacing kernel versions of functions with user space versions as well as making installation scripts for each interconnected system. These previous projects either do not provide instructions for replication or the solution developed is incomplete.

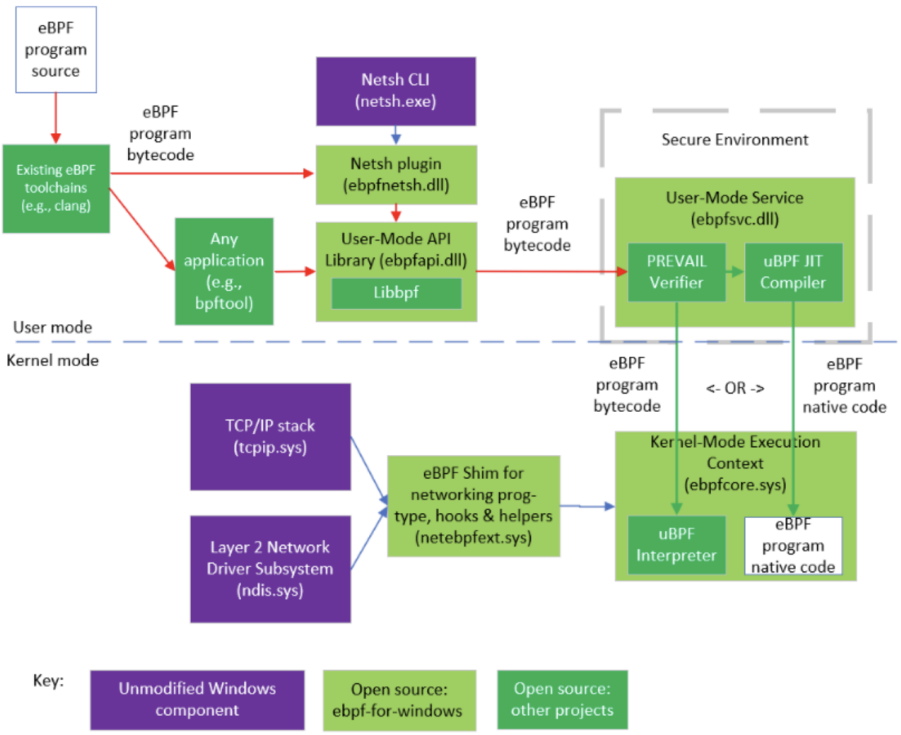


Fig. 9: An overview of the eBPF-for-Windows ecosystem [19]. The target software for our fuzzing harness, the PREVAIL verifier and uBPF virtual machine, are in the top-right of the figure, run in a secure environment in user space.

To test our fuzzing harness we deploy it against a verifier and virtual machine. We chose the PREVAIL verifier and uBPF virtual machine. These are used as part of the eBPF-for-Windows project [19], developed by Microsoft to bring

the eBPF technology to the Windows platform. An overview of the project is shown in Figure 9. The PREVAIL verifier and uBPF virtual machine are to be run in user space, in a secure environment. Both PREVAIL and uBPF are implemented according to the eBPF specification, meaning that they should agree on the behavior of an eBPF program. The targets are also simple to run in user space, meaning that faults are easier to observe, compared to programs run in kernel space.

4.3.1 PREVAIL PREVAIL (Polynomial-Runtime EBPF Verifier using an Abstract Interpretation Layer) is a verifier developed as an alternative to the Linux verifier. PREVAIL runs in user space, but also contains an interface for calling the Linux in-kernel verifier. PREVAIL was developed with the aim of overcoming the limitations of the Linux verifier with regards to efficiency and formal correctness. It uses a different architecture than the Linux verifier to perform verification, that is able to verify eBPF programs faster. The architecture of PREVAIL is designed to overcome:

1. Reduce answers that are false negatives.
2. Scale to programs with a large number of paths.
3. Support programs with loops.
4. Add formal correctness to the verification.

The formal model behind PREVAIL uses the language eBPFPL; a core low-level programming language, used to capture the essence of an eBPF program, but also applicable to other kernel extensions. The operational semantics of PREVAIL enforces safety at runtime by aborting into an error state when a safety violation is detected. A static analysis is used to prove the safety of a program, over-approximating the semantics. If the implemented analyzer reaches the conclusion that the error state is never entered, the program can be concluded as safe to execute.

The PREVAIL verifier uses Zone Crab, a relational abstraction domain for a Control Flow Graph (CFG) based language, CrabIR, whereas the Linux verifier uses CFG validation, to check unreachable instructions and execution paths for program termination.

However the PREVAIL architecture exceeds memory usage deemed practical for use as an in-kernel verifier. In addition to safety of execution, there are also certain special checks that is performed by the Linux verifier that were not implemented in PREVAIL.

4.3.2 uBPF uBPF is a user space virtual machine that supports JIT compiling eBPF programs into eBPF native machine code. The compiled eBPF programs can then be run on the uBPF virtual machine. uBPF facilitates compiling eBPF programs without accessing the Linux kernel eBPF JIT compiler and running the resulting machine code instructions independently of the host machine. Errors in the kernel and host machine could result in crashes or data corruption. Running eBPF programs in user space can be useful in context of testing and debugging eBPF programs, as errors are reported more accessibly.

4.3.3 Target Input eBPF programs intended for the Linux kernel are often written in C, using the eBPF library, and compiled using tools like clang, as mentioned in Section 2.2. The generated bytecode is stored in an ELF file and passed to the Linux kernel. It is therefore common for eBPF technologies to implement an ELF file parser. This makes ELF files a suitable format to generate for, when generating eBPF programs.

Both PREVAIL and the uBPF virtual machine accepts ELF object files as an input. While PREVAIL and uBPF also accept human readable instructions, as presented in Example 4, generating ELF files as inputs is the more flexible format, as this is accepted by most eBPF technologies. As ELF files are written in bytecode, the generated eBPF program has to be bytecode as well.

Producing ELF files as target input greatly increases reproducibility and documentation, as the files can easily be stored, for later target inspection. Execution of ELF files produce no side effects that need to be reset or accounted for meaning that targets can easily be rerun with a given ELF file. ELF files also result in great reusability as it is the shared input format.

4.4 Generate Fuzz Data

In the following sections we will describe our approach to generating fuzz data. In Section 4.4.1, we will present the chosen fuzzing methodologies. In Section 4.4.2, we present an overview of the generator component of our fuzzing harness. In Section 4.4.3, we present the Rust crate, rBPF, used in the generator component to generate eBPF bytecode. In Section 4.4.4, we present *strategies* deployed by our generator, to generate instructions performing specific eBPF program behavior In Section 4.4.5, we present the parser component that produces an ELF file with the eBPF bytecode using the Faerie Rust crate.

4.4.1 Fuzzing Approach In the following sections we descibe our fuzzing approach, i.e. black-, grey-, or whitebox fuzzing, as well as our approach to generating fuzz data. Our approach is chosen as we aim to generate programs that are parsed and unmarshaled correctly by the targets.

Blackbox Fuzzer We choose to develop a blackbox fuzzer. Our reasons are mostly to address resource constraints. One reason is the relative ease of implementation, compared to grey-, or whitebox fuzzing. Another reason is due to our design goal of component modularity. Designing a whitebox fuzzer for our targets would limit the use of our fuzzer to these targets. The same reason can be applied to greybox fuzzing, as different tools would have to be instrumented differently. Therefore, to create a tool which is simple to set up for different eBPF technologies, we choose to develop a blackbox fuzzer.

Generation-Based We will develop a generation-based fuzzer. The approach taken to produce fuzz data will be a generator that generates eBPF programs

instruction by instruction. The intention is that the structure provided by generating instructions, instead of mutating bytecode, will be more likely to be verified as safe. This should result in more eBPF features being covered during the execution of generated programs. Features are tested by generating instructions that are correct or incorrect according to the bounds derived from the eBPF specification, e.g. reading from uninitialized registers.

Smart Fuzzer The fuzzer should greatly benefit from being protocol aware by generating and saving data in a structured manner, instead of mutating input files. Parsing eBPF programs to an ELF file requires correct section linking, for the ELF file to be recognized as containing an eBPF program. This is implemented in our harness. Additionally, the generation of eBPF programs is partly based on the specification and structures derived from the eBPF input format. As we generate eBPF instructions, and produce an ELF file containing the resulting bytecode, we classify our fuzzer as a smart fuzzer

Unguided Fuzzer We design in unguided blackbox fuzzer. This means that the only knowledge utilized by our fuzzer is the specifications derived from the eBPF input format and no external target information is used.

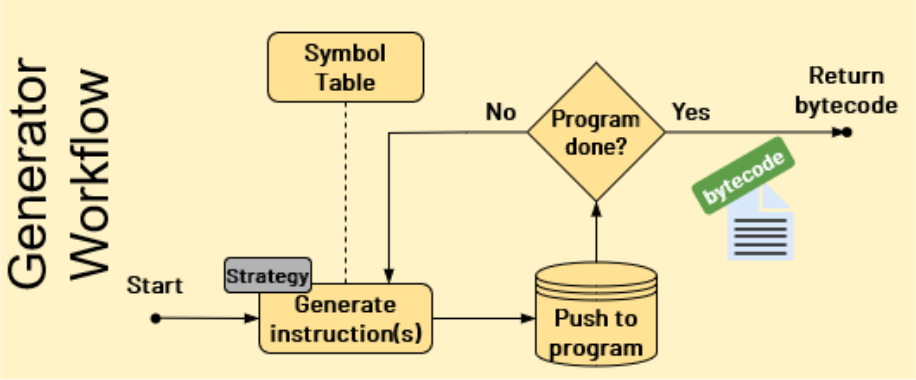


Fig. 10: An overview of the workflow of the generator component. Instructions are generated and pushed to a program struct. When enough instructions have been generated the bytecode is returned.

4.4.2 Generator Component We have chosen smart generation-based blackbox fuzzing approach. The target input is eBPF bytecode written in the ELF file format. In the following section, we describe a generator designed to generate eBPF bytecode. An overview of the general workflow of the generator component is shown in Figure 10. The main functionaility of the generator revolves

around *strategies*. Depending on the strategy, the generator will utilize different methods to generate eBPF programs, e.g. selecting random instructions, or generating structures resembling stack utilization. These instructions are pushed to an eBPF program struct containing each generated instruction. If a strategy requires knowledge of eBPF semantics or previously generated instructions, information can be found in a symbol table. When a set amount of instructions have been generated, the struct is passed to the parser component. To support this structured approach to input generation, we use the Rust crate, rBPF [24], which provides an interface for creating low level eBPF program instructions.

4.4.3 rBPF rBPF, inspired by uBPF [14], is a Rust based project that implements a virtual machine for compiling and running eBPF programs. rBPF contains an interpreter, an x86_64 JIT-compiler, and a disassembler. We use rBPF as it provides a high level interface to low level eBPF program generation through registers and instructions. rBPF provides methods and structures that facilitate writing eBPF instructions easily, as presented in Example 8. This ensures correct eBPF instructions both in bytecode form and human-readable form making eBPF program generation simple.

Example 8 (rBPF Instructions).

Consider Listing 3 and the corresponding bytecode produced when compiling the C code. The Rust code, utilizing the rBPF crate, to produce the first two bytecode instructions, i.e. move and store, are presented in Listing 6. A struct variable `prog` stores the current instructions pushed to the program. All values in the `set` functions can be written as either decimal or hexidecimal.

```
prog.mov(Source::Imm, Arch::X64)
  .set_dst(1)
  .set_imm(40)
  .push();
prog.store_x(MemSize::Word)
  .set_dst(10)
  .set_src(1)
  .set_off(-4)
  .push();
```

Listing 6: Rust code utilizing the rBPF crate to write the first two bytecode instructions produced when compiling the C code in Listing 3.

4.4.4 Generator Strategies If the generator simply produced random bytecode, the probability of a generated program being a valid eBPF program and passing verification would be very low. The generator should produce syntactically valid programs while still applying meaningful randomness for fuzz testing the verifier and virtual machine components of the fuzzing harness.

The rBPF crate ensures that the generated eBPF instruction syntax is correct. To further increase the proportion of valid programs some semantics of

the eBPF programs should also be correct. For example when the generator has to load a value to the stack, it principally uses register 10, as this is the stack pointer. For fuzz testing purposes, the generator is designed to be flexible in its use of semantic rules, such that use of incorrect registers is also tested.

To adhere to semantic rules we draw inspiration from Csmith, described in Section 7.3, a blackbox fuzzing tool for C compilers for generating random C programs. We use a symbol table to keep track of certain information regarding the generated eBPF program, e.g. which registers have been initialized. The symbol table can also be used to account for eBPF limitations, such as available registers.

The generator produces instructions using various functions covering certain types of instructions. We denote a sequence of these functions as a *strategy*, as they can be combined to generate and target certain eBPF program features. In the following sections, we describe the generation strategies designed for the generator. The aim of these strategies is to cover as much of the eBPF technology as possible, e.g. targeting maps or stack operations.

A strategy targeting stack operations might focus on testing whether verifiers can catch memory access outside the stack memory bounds. This is also called an out of bounds access, which is a type of unspecified behavior [12]. The aim is then for the strategies to cover as many undefined and unspecified behaviors within the eBPF technology as possible.

A strategy aiming at testing helper functions could be ideal, but such a strategy can not be implemented, because uBPF has limited support for helper functions. Three eBPF helper functions exist to create, look up, and delete elements from maps. This means uBPF has not implemented user space versions of the remaining helper functions supported in kernel eBPF.

Random Instructions A baseline strategy is designed to select random instructions performing random operations. The registers, offset, and immediate values chosen for these instructions are selected at random as well. We add a few flags, such that registers and similar are selected both within accepted ranges, but also outside, e.g. registers above 10.

Stack Instructions As eBPF programs are limited to ten registers the 512 byte stack is utilized to store values. eBPF programs utilize the stack to store values for preserving values across function calls, thus freeing the limited amount of registers for further use. Programs often load values into available registers, then apply some operations on those values, before pushing the results to the stack. After the function call, values can be loaded from the stack, and further operations are performed. This pattern repeats as needed throughout the eBPF program. An example of this pattern can be seen in Example 4, where values are pushed to the stack, then popped, and finally an operation is performed on the values.

Generating this structure by randomly choosing instructions would be very unlikely. Therefore we design functions, to generate sequences of instructions

which resemble a program utilizing the stack. This is primarily done by tracking the stack height. Knowing the stack height, instructions can be generated pushing values to the top of the stack, or popping a value from the stack. For fuzz testing using this strategy, we add some randomness such that instructions can be generated that write values anywhere on the stack, possibly overwriting previous values. We also add the possibility of attempting to write outside of the stack boundaries.

Scannell Maps One of the fuzzers developed for the eBPF technology applies a specific strategy that targets the maps feature of eBPF programs. We design a strategy that aims to replicate the out of bounds write resulting from this strategy, decribed in Section 7.1.

The strategy revolves around performing a random number of ALU and branching operations involving a register pointing to a map. It then attempts to save a value to the fuzzed map pointer. If nothing was written to the eBPF map when checking afterwards, then the value was written outside the map memory bound.

This out of bounds write should not be possible and should be caught during verification of the given eBPF program. The purpose of this strategy is to test whether the faulty pointer arithematic can be replicated in the user space targets.

Random Maps This strategy aims to target the maps allocation methods used in the eBPF virtual machine. The map is initialized by generating random bytes for each attribute of the eBPF map, which covers map type, key size, value size and max entries. There are additional attributes which are optional that can also be generated, such as map flags, inner index and noma nodes. The strategy revolves around preparing the stack for an eBPF map lookup and then trying to read from the randomly generated map.

Rule Breaking Instructions We design a few functions that purposefully attempt to break verification or semantic rules of eBPF. These functions can be appended to other strategies, such that the strategies are more likely to contain a sequence of instructions with undefined or unwanted behavior.

Examples of rule breaking sequences could be the creation of infinite loops by generating jump instructions with specific values. Registers could also be used incorrectly. This includes writing to the stack pointer, and generating a high number of ALU instructions in a row. Generating a large number of ALU instruction could break certains counts that are used to enforce the rules presented in Section 2.5.3. An idea also used in other eBPF fuzzers, as presented in Section 7.1.

4.4.5 Parser Component

The generated program is passed to the parser component, which handles the creation of the ELF file. To produce an ELF file with our generated eBPF program, we use the Rust crate, Faerie. By passing

the generated bytecode and an architecture struct Faerie can generate the ELF file. An overview of this work flow is presented in Figure 11. By specifying a target, i.e. eBPF, and an OS architecture, Faerie handles the creation of the relevant magic bytes in the ELF file header. A few links for sections in the ELF file have to be defined as well. This includes a section for the eBPF program itself, and sections defining the maps used by the eBPF program. The eBPF program section must be named `.text`, and map sections are named `maps`.

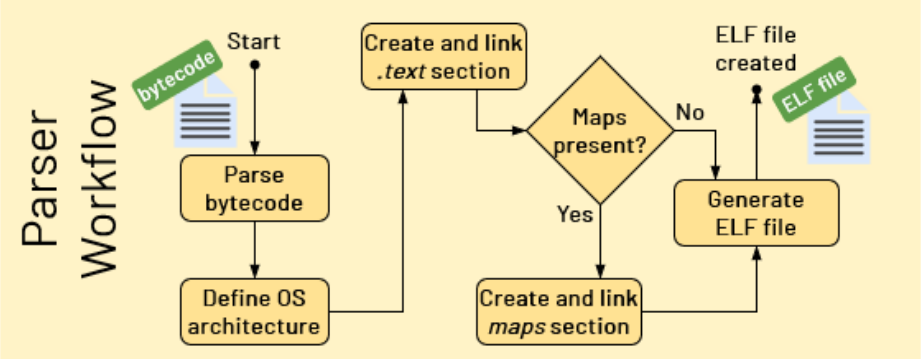


Fig. 11: Workflow of the parser component. The generated bytecode is parsed to a format, which can later be written to the ELF file. The Rust crate, Faerie, handles creation of sections, section links, and magic header bytes by passing a struct containing the architecture information.

Faerie The Faerie crate is not actively maintained, so new features do not get released often. A developer on Github, Kitlith, has a working implementation that facilitates setting arbitrary permission flags for ELF sections, which are required for eBPF `.text` sections, i.e. the section containing the eBPF program. We create a forked version of Faerie that contains the arbitrary permissions implementation, an updated ELF target interface that supports eBPF and an updated data section that support eBPF maps.

4.5 Execute Fuzz Data

When an ELF file containing the eBPF bytecode has been produced it can be unmarshaled by the PREVAIL verifier. PREVAIL then determines whether the program is safe to run. Safe programs are then run by the uBPF virtual machine. Both PREVAIL and uBPF are compiled locally and can be called with ELF files as arguments.

As standard output, PREVAIL returns three numbers, or an error if the program could not be unmarshaled correctly. Errors shortly describe the first observed broken semantic rule, e.g. incorrect use of a register, or a jump out of bounds. The three returned numbers are:

1. **Safety**: 1 if the eBPF program is safe to run, 0 if not.
2. **Execution Time**: Time in seconds spent verifying the eBPF bytecode.
3. **Memory Usage**: The maximum number of bytes used during execution.

When uBPF has executed an eBPF program, the standard output is the contents of register 0, as this is the output of an eBPF program. An error message may by returned, if something unexpected happened during execution. Before executing the eBPF program, uBPF also does some basic safety checks, e.g. checking for loops in the program.

4.6 Fault Detection

Our approach to fault detection is checking the output of the fuzz targets. If PREVAIL verifies a program as safe, the program is run in the uBPF virtual machine. The return value of an eBPF program is the contents of register 0. Part of the verification process is determining if register zero is initialized, and thereby has a return value.

When a program is run in the uBPF virtual machine without returning anything in register 0, a fault must exist in either PREVAIL or uBPF. As neither target can be considered an oracle, it can not be immediately determined which target contains the fault. Further manual inspection is needed to determine where the nonconformance lies.

For strategies revolving around manipulating maps, the generated eBPF programs will mostly have a deterministic setup. This is to ensure correct use of the maps feature and that errors can be caught. When an attempt is made to write a value to the map, a certain value can be chosen, e.g. 42. Determining whether the given value was written correctly to the map or out of bounds, is done by reading from the map to register 0. When uBPF finishes executing it is checked whether the value 42 is returned. If not, the value was written out of bounds.

5 Implementation

In the following chapter we present implementation details of our fuzzing harness, and focus especially on the generator and parser component. The first few sections are intended to provide the information needed to replicate our fuzzing results. This includes setting up the code from our GitHub repository, and installing necessary tools and dependencies. In the second part of the chapter we present certain implementation details of our fuzzing harness components. We have chosen certain snippets of code, to give an idea of the supported features of our harness. The harness itself is implemented in the Rust programming language, but code snippets are presented in a pseudocode format for a more readable presentation.

Chapter Outline

In Section 5.1, we present the structure of out code repository, and provide information such that the target software can be build. In Section 5.2, we present how our fuzzing harness can be installed on run. In Section 5.3, we present the general structure of a fuzz target, and how strategies are set up and executed. In Section 5.4, we present a few functions implemented in our symbol table. In Section 5.5, we present implementation details of the different strategies in the generator. In Section 5.6, we present how the generated eBPF program is parsed to an ELF file.

5.1 Code Repository Structure

In this section we present an overview of the codebase for our project. The focus will be on the structure of the codebase and the details of the used submodules. This should serve as part of a guide on how to replicate our results, along with Section 5.2 and Section 5.3.

Our code repository can be found at:

`https://github.com/m-tolstrup/buzzy`

The codebase consists of six folders, four containing submodules used by the fuzzer:

- `ebpf-verifier`: The PREVAIL eBPF verifier repository.
- `ubpf`: The uBPF virtual machine repository.
- `rbpf`: The repository for the Rust eBPF project.
- `buzzy`: The code for our fuzzing harness, written in Rust, combining the tools provided by the submodules.
- `buzzy/faerie`: This submodule provides methods for producing ELF files.
- `scripts`: Different Python scripts for automatization.

5.1.1 Submodules Below we present an overview of the submodule resources that are used to facilitate our fuzzing harness. This setup was tested on a Linux machine running the Linux Mint 21.1 Cinnamon OS. Running our fuzzing harness is covered in Section 5.2. Our scripts can be run as usual Python scripts.

To clone our repository, including the necessary submodules, the following git command can be used:

```
git clone --recurse-submodules https://github.com/m-tolstrup/buzzy
```

All submodules can be updated to point at the specified branches by:

```
git submodule update --remote <repository>
```

While the project has focused on targeting PREVAIL and uBPF, other eBPF verifiers or virtual machines could be targeted as well. No part of the fuzz harness is implemented specifically for PREVAIL or uBPF.

ebpf-verifier A guide for compiling the PREVAIL verifier can be found on the PREVAIL GitHub page [38]. PREVAIL provides eBPF samples that can be used to test whether the verifier was compiled successfully. The PREVAIL project also provides an interface to use the kernel eBPF verifier, by using the flag `--domain=linux`. The in-kernel eBPF verifier checks for infinite loops by default. To enable this check in the PREVAIL verifier, an additional flag `--termination` has to be used.

uBPF A guide for compiling the uBPF project can be found on the uBPF GitHub page [14]. uBPF is in active developement and there are some features which have yet to be implemented or released. There are some features that are partially implemented, such as eBPF maps support and helper functions. uBPF only supports one eBPF maps type, arrays. The function call mapping differs between eBPF and uBPF, as a limited set of helper functions have been implemented such as those relevant to eBPF maps.

rBPF After downloading our repository and its submodules, no further installation is needed for the rBPF submodule. The methods provided by the submodule are used in our fuzzing harness, and are compiled and executed when running our harness.

Faerie As described in Section 4.4.5, The Faerie crate is not actively maintained. To facilitate the necessary features, we have created a fork of Faerie which implements the requirements for eBPF.

The sections used for the eBPF programs require certain flags to be specified such that they are recognized by PREVAIL, uBPF, and eBPF technologies in general. The current version of Faerie assumes a set of default flags for each type of ELF section. A developer on Github, Kitlith, has a working implementation that supports arbitrarily setting these section flags. We merge this implementation into our fork.

Faerie uses a library called target-lexicon, which facilitates specifying the architecture that the ELF file format is for. Faerie currently uses target-lexicon version 0.12, whereas eBPF support was added in a later version 0.12.2. We update our Faerie fork to use the later version.

eBPF maps are expected to be defined in a data section called `maps` in the ELF file. Faerie prepends the name of data sections with either `.data.` or `.rotdata.`, as data sections are usually used for global variables or constant strings. We change Faerie such that when a data section has the name `maps`, nothing will be prepended.

5.2 Rust Fuzz Testing Setup

In the following section we describe the tools used to provide and run our fuzzing harness. We also provide the necessary steps to set these up. These tools have been set up and tested on machines running the Linux Mint 21.1 Cinnamon OS.

Our fuzzing harness is implemented in the Rust [37] programming language. The different releases of Rust can each be installed as a toolchain supporting a stable, beta and nightly version of the Rust compiler.

The recommended tool to use for fuzzing in Rust is cargo-fuzz [28]. The cargo-fuzz tool manages the fuzzing of a fuzz target and uses the LLVM project libFuzzer [17] fuzzing tool. LibFuzzer can use sanitizer flags meaning the nightly Rust compiler is required, as they are classified as experimental features. We use cargo-fuzz, but the implementation ended up mostly relying on the fuzzing iteration engine.

To utilize the public libraries that support fuzzing in Rust, we use the rustup installer to install the *nightly* toolchain. The Rust nightly release provides easy access to the nightly Rust compiler and the experimental in-development features it supports. The nightly release can be installed using rustup from the command line as follows:

```
rustup cargo install nightly
```

The cargo-fuzz tool and the Rust crate for the LLVM libFuzzer (libfuzzer-sys) can be installed as follows:

```
cargo install cargo-fuzz
```

With the tool installed, a project can be set up for fuzz testing as follows:

```
cargo fuzz init
cargo fuzz add <target>
```

This creates a folder structure for fuzz targets, and a Rust file for the defined target. Note that the `init` command has already been run for our project, but new targets can be added. The `<target>` is the name of the Rust file, without the `.rs` file extension. In the created Rust file, users write code to set up the target function(s). Multiple targets can be added and fuzz tested, if different setups for the target is wanted. A fuzz test can be run on the target as follows:

```
cargo +nightly fuzz run <target>
```

The code we have written for the fuzz targets can be found at the following path:

```
<path-to-project>/buzzy/fuzz/fuzz-targets/<target.rs>
```

Adding and running the fuzz targets should be done in the `buzzy` folder.

5.3 Fuzzing Harness Structure

In the following section we present how we set up our fuzzing targets. For each *strategy* deployed by our fuzzer we have designed a fuzz target using:

```
cargo fuzz add <target>
```

The targets follow a general structure but differ in some areas. When a target has been added, the cargo-fuzz crate creates a file named `<target>.rs`. Inside the target file, a function named `fuzz_target!()` can be found. In this function code should be set up to generate fuzz data, pass it to the targets, and log faults if any are detected.

Code for the symbol table, generator, and parser can be found at the following path:

```
<path-to-project>/buzzy/src/<component.rs>
```

Pseudocode for our general approach is presented in Algorithm 1. The implemented generator is called to generate a random eBPF program. The generator is called with a strategy which determines how the generator generates the eBPF program. The generated program has to be parsed to an ELF object file. When the ELF file has been produced, the PREVAIL verifier can be called and the result is stored in a variable. If the PREVAIL verifier marks the program as safe, we pass it to the uBPF virtual machine. If uBPF does not terminate by returning a value in register 0, we log the generated program, as this should not be possible.

Algorithm 1 An overview of our general approach of implementing the `fuzz_target!()` function.

Input: —
Output: —

```
1: while time_left() > 0 do
2:     strategy ← "Random"
3:     generated_prog ← generate_program(strategy)
4:     parsed_prog ← parse(generated_prog)
5:     path ← "../obj-files/data.o"
6:     write(parsed_prog, path)
7:     verifier_result ← prevail_verify(path)
8:     if is_safe(verifier_result) then
9:         vm_result ← ubpf_jit(path)
10:        if is_reg_zero_empty(vm_result) then
11:            log(generated_prog)
12:        end if
13:    end if
14: end while
```

5.4 Symbol Table

We use a symbol table to keep track of certain features of the generated eBPF program, and eBPF semantic rules. This includes keeping track of register use,

current stack height, and generation of random numbers. In this section, we present a few functions, provided by the symbol table, used to provide structure to generated eBPF programs.

5.4.1 Tracking Registers We have implemented a set of flags, variables, and functions to track register use. In Algorithm 2, a function used to select a random register for load instructions is presented. The function checks if the array `stored_registers` is empty. This array tracks registers last used in store operations on the stack, meaning the value of the register has been saved on the stack. If the value of the register has been saved, it is a fitting candidate to load new values into. The function uses registers 0 through 5, as these are the return register and the function call argument registers. The function also uses a flag, `all_registers_allowed`, to determine whether it should use all the available registers. This can be enabled for testing purposes.

Algorithm 2 A function returning a random register, when generating a load instruction.

Input: —
Output: A random register, `reg`.

```
1: reg ← ⊥
2: if stored_registers.is_empty() then
3:     if all_registers_allowed then
4:         reg ← random_range(0..10)
5:     else
6:         reg ← random_range(0..5)
7:     end if
8: else
9:     reg ← stored_registers.pop()
10: end if
11: loaded_registers.push(reg)
12: return reg
```

5.4.2 Stack Functionality Some strategies in the generator focus on the eBPF program stack. To generate programs that utilize the stack correctly, we use an integer value, `stack_height`, in the symbol table that tracks the amount of bytes stored on the stack. Whenever a byte, word, etc., is pushed or popped from the stack, the `stack_height` variable is adjusted by the corresponding amount of bytes. Tracking this allows generator functions to generate instructions close to what they would look like a correct eBPF program. Stack functionality is then provided through different functions in the symbol table:

`stack_push`: Add the number of bytes pushed to the stack to `stack_height`.
`stack_pop`: Subtract the number of bytes popped from the stack from `stack_height`.

`stack_top`: Return the value of `stack_height` subtracted from the maximum stack capacity (default is 512 bytes).
`stack_bottom`: Return the current value of `stack_pointer`.

The functions `stack_top` and `stack_bottom` can be used to generate instructions that try to push values at the top and bottom of the stack. By generating large enough offsets, instructions can be generated that attempt to store and load outside the stack boundaries as well.

5.5 Generator Component

In the following section we present implementation details of the generator. We first provide a general overview, followed by implementation details specific to the different generator strategies.

The instructions generated in the generator all follow big-endian notation and a 64-bit architecture.

5.5.1 Structure Overview In Algorithm 3, we present the function called at Line 3 in Algorithm 1. The `generate_program()` function fills a struct named `prog` with instructions. The chosen instructions are based on the strategy passed to the generator. The functions called in the different branches of the switch-statement can be reused and combined to create new strategies.

Algorithm 3 The function in the generator handling what instructions to generate. A strategy is passed to the generator, and depending on the strategy, different functions are called, adding instructions to the `prog` struct.

Input: A string, `strategy`.
Output: A struct containing an eBPF program, `prog`.

```
1: prog ← BpfCode.new()
2: switch strategy do
3:     case "Random"
4:         prog.append(generate_random_instructions())
5:     case "StackOpSequences"
6:         prog.append(generate_stack_seq())
7:     case "ScannellMaps"
8:         prog.append(generate_map_header())
9:         prog.append(generate_map_body())
10:        prog.append(generate_map_footer())
11:    case ...                                    ▷ More strategies
12:        ...
13: return prog
```

5.5.2 Generator Strategies In the following sections we present implementation details specific to each generator strategy.

Random Instructions In Algorithm 4, we present the function called at Line 4, in Algorithm 3. This function is used to generate a set amount of random eBPF instructions. Multiple strategies use a setup with an instruction count loop, and uses weights to determine what instructions to generate.

A random number is generated to determine what class of operation should be generated, e.g. ALU, or store operations. The probability of generating a certain class of operation can be changed by adjusting the numbers in the switch cases and the generated random number. After an operation class has been selected, an operation within the class is chosen, and registers, offset-, and immediate values are generated for the instruction. Flags are set to determine whether these values should be generated within legal boundaries.

The number of instructions to generate, i.e. `instr_count`, is determined by the symbol table. This number can be set manually or randomly generated.

Algorithm 4 A function implementing the strategy for generating random instructions. Depending on the ranges of numbers for the switch cases, the probability of generating certain eBPF instructions can be changed.

Input: —
Output: Generated instructions, `instr`.

```
 1: instr ← ⊥
 2: while symbol_table.instr_count > 0 do
 3:     rand ← random.range(1..10)
 4:     switch rand do
 5:         case rand.in_range(1..2)
 6:             instr.append(generate_alu_instr())
 7:         case rand.in_range(3..6)
 8:             instr.append(generate_store_instr())
 9:         case ...                                  ▷ More functions
10:             ...
11:     symbol_table.instr_count −= 1
12: end while
13: return instr
```

Stack Instructions In addition to the stack functionality provided by the symbol table, described in Section 5.4, the generator has functionality for structuring sequential load and store operations. When generating load and store instructions, the generator functions select a random memory size, e.g. byte or word, and generates a sequence of instructions. As each instruction is generated, the `stack_height` variable is updated accordingly. Registers can be selected in

sequence as well, to avoid that popped stack values are loaded into the same register. Some noise has a chance to be added, such that operations are not always performed at the exact stack height. The fuzzer can generate instructions that attempt to write outside stack boundaries as well. Between stack operations, this strategy generates jump and ALU instructions, where the registers used in stack operations are chosen for destination and source registers.

Scannell Maps To implement the Scannell maps strategy we replicate the three step structure described in Section 7.1. The generator generates and appends each step, the header, body and footer to the `prog` struct, as seen in Algorithm 3.

The header has functionality for preparing the stack for a map memory lookup. The map used in the lookup is defined in the ELF file, with attribute type as array (2), key size as 4 bytes, value size as 8192 bytes and max entries as 1. This is the map specification used by Scannell.

When generating the header a word (the key size) sized load from the map is prepared first. This is done by storing a word on the stack, and pointing register 2 to the stack with an offset of 4 bytes, the current stack height. The helper function `map_lookup_elem`, is then called through a call instruction with an immediate value set to its ID. This call instruction sets the destination for the map pointer as register 0, and utilizes the map file descriptor and address expected at register 1 and register 2. The map pointer is then verified by adding a conditional jump instruction, which exits the program if the value at register 0 is still its initial value of 0. The function call resets registers 1 to 5, so the header then initializes the registers, 1 and 2, by reading from the map.

The body reuses the functions for generating random ALU and jump instructions, but the instructions are generated using only register 1 and 2. Each instruction generated will randomly use one of the registers as the source and the other as the destination.

The footer then ensures that a memory write is actually performed with the map pointer. Before attempting to write to the map, register 1 or 2 is subtracted or added to an arbitrary map pointer register, 4. A store operation is then generated using register 4, to utilize the randomly generated pointer. Lastly, the footer ensures that the eBPF program has a valid return value by moving the value 1, to register 0.

The conditional jump used for verifying the map checks whether register 0 was set to the map pointer, by ensuring that it does not contain its initial value. This check fails as PREVAIL asserts that conditional jumps can only check a value, not a map pointer.

Additionally, PREVAIL verifies the header as valid without the map verification, but the map lookup function used in the header causes a segmentation fault when executed in uBPF.

Every attempt to pass the generated programs to uBPF resulted in a segmentation fault. The segmentation fault is described in Section 6.3.4. Maps support has recently been added to uBPF, and the implementation is still in development.

Random Maps This strategy reuses the function for generating the header from the Scannell maps implementation. The map used for the lookup is instead defined randomly, by setting each attribute to random bytes.

Only the header is used, such that the generated map is verified and accessed. The generated map lookup call instruction and conditional jump will verify that the random map can be correctly initialized. Two instructions are generated to initialize two new registers by reading from the map. The generated memory read instructions will verify that new registers can be initialized using values from the map.

Rule Breaking Different functions have been implemented to purposefully break rules, or create unwanted or undefined behavior. Instructions are allowed to write to the stack pointer, by selecting register 10 when the given instruction is generated. Infinite loops are created either by creating jumps negating the program counter, or using a variable to track where a previous jump occurred, resulting in instructions jumping back and forth between each other. To create a large number of ALU instructions a wrapper function is created, which calls the function generating random ALU instructions a certain number of times.

5.6 Parser Component

The PREVAIL verifier and the uBPF virtual machine looks for an eBPF program in a section named .text in the input ELF file. We produce an ELF file with the given generated eBPF program. This is done through following steps:

1. Create the object file at a defined path.
2. Initialize an ELF artifact with a target specification.
3. Create each ELF section declarations.
4. Create relocations.
5. Produce ELF file.

The Rust standard library is used to create and write to a file. To produce an ELF file we use the Faerie crate, which implements methods to define the necessary components of the ELF file. Faerie facilitates defining the target specifications of the ELF file, which is used to generate the required magic bytes for the eBPF architecture, Linux OS, and ELF binary format in the ELF header.

ELF file sections and relocations can be created using the Faerie `ArtifactBuilder`. The `ArtifactBuilder` has to be initialized with the target specifications struct. After initializing the `ArtifactBuilder` section declarations and their definitions can be created.

A section named `.text` is declared, as this is the section name conventionally used when creating ELF files for eBPF programs. The generated eBPF program is translated, using the built in `into_bytes()` function from rBPF. The `.text` section is then defined to contain the eBPF program instructions byte code. The `.text` section is only recognized when marked with the executable and allocatable ELF section flags.

When an eBPF map is required, a section named `maps` is declared. This is the current maps section naming convention, however it is slowly being standardized as `.maps`. The `maps` section is then defined to contain the necessary bytes for each eBPF maps attribute, map type, key size, value size and max entries. The `maps` section is only recognized when marked with the writable and allocatable ELF section flags.

The `ArtifactBuilder` facilitates relocations, i.e. linking from one declaration symbol into another. For a `maps` section to be recognized as in use, a symbol has to point into the maps section via a relocation. A link is created from a specific offset pointing to a load instruction from the `.text` section to the `maps` section. This relocation determines the eBPF map address, from which the load instruction loads.

Using the `ArtifactBuilder`, the ELF file is produced and written to the object file, and the file is ready to be verified.

5.7 Target Execution and Fault Detection

Target Execution Calling the targets is done by using the Rust standard library. The Rust standard library contains a `process::Command` module allowing terminal commands to be run programmatically. Having compiled PREVAIL and uBPF these can be called with the generated ELF file. We run PREVAIL with the flag `--termination` flag, to check against infinite loops. uBPF is run with the `-j` flag determining that JIT-compiling is used instead of interpretation.

Fault Detection Fault detection is determined by checking the standard output of uBPF. If this is not a 64 bit value, i.e. the contents of register 0, an error must be present as PREVAIL has verified the program as safe. When this is detected, the ELF file containing the eBPF bytecode is stored in a log folder for later inspection.

Some segmentation faults or similar cause the fuzzing harness to terminate, given the implementation of the standard library used for executing the targets. In these cases, the ELF file generated is not saved in the log folder, as this is the last step of a fuzzing iteration. The ELF file causing the segmentation fault can then be found in the `obj-files` folder; the folder where the produced ELF files are stored and read by the fuzz targets.

6 Evaluation

In this chapter we evaluate our fuzzing harness. This is done by conducting a set of experiments testing the ratio of valid and invalid programs. The results are then assessed in two parts. First we assess the effectiveness of the strategies with regards to the ratio of valid and invalid programs. Then we assess the found bugs, and discuss the placement of these bugs in PREVAIL and uBPF. Based on these assessments we will discuss and conclude on our design goals.

Chapter Outline

In Section 6.1, we present information regarding our experimentation setup. In Section 6.2, we present experiments where the implemented strategies are tested and compared by their ratio of valid and invalid program. We also discuss the frequency of how often specific strategies uncover certain bugs. In Section 6.3, we present the bugs found with our fuzzing harness. In Section 6.4, we discuss and assess our chosen design goals and approach.

6.1 Experiment Setup

To determine the effectiveness of the implemented generation strategies we run a set of experiments. We consider effectiveness regarding the areas of eBPF program validity and the strategies ability to target features.

We run the different experiments for one hour in order to collect data on the ratio of safe programs generated. We focus on gathering data on the safety of generated programs as this fuzzing phase was in focus during design and implementation. We run the experiments on a computer running the Linux Mint 21.1 Cinnamon OS. As we consider the ratio of valid programs and not the total amount generated, the hardware of the computer does not impact experiment reproducibility.

The saved data is aggregated with different Python scripts, providing ratios between the number of valid and invalid programs. All scripts used during the project are available in our code repository.

We also run the random instruction and stack instruction strategies for 24 hours. We do this in order to see if more bugs can be uncovered.

Result Format PREVAIL provides three data points when it is used to verify a program. It provides information on program safety, time spent verifying, and memory used during verification. If an error occurred during unmarshaling of the ELF file, this is reported as well. Based on this information, we measure and classify generated programs as follows:

- **Valid programs**: Programs that pass the unmarshaling state of PREVAIL and are considered safe to run.
- **Invalid programs**: Programs that pass the unmarshaling state in PREVAIL, but are not considered safe to run.
- **Erroneous programs**: Programs that do not pass the unmarshaling state of PREVAIL, e.g. because of syntax errors.
- **Parsing failed**: Some programs cause errors in our parser component.

When recording data for the experiments we also collect data on the length of the generated program, i.e. how many instructions were generated. This allows us to create graphs plotting the length of the programs against the amount of safe or unsafe programs to see how this is related.

When programs are parsed to ELF files by Faerie, an error is encountered where Faerie is unable to produce the given ELF file. We could not find a solution for this in time.

6.2 Experiments

We devise a set of experiments that will determine the effectiveness of each generation strategy with regards to the program validity. Through these experiments we want to assert that structured strategy design positively affects the ratio between valid and invalid eBPF programs. Furthermore, we also assert that it is possible to target specific eBPF features with fuzzing strategies, by observing the frequencies of a given bug being produced. We will evaluate each experiment by itself, and summarize the results in Section 6.2.4 to conclude on the effectiveness of strategy based generation. These results are one of two parts of our evaluation of the strategies.

All random choices are equally distributed in the run experiments. An early observation showed that longer eBPF programs are rarely verified. For all experiments, we therefore add weights to make programs shorter than 32 instructions appear more often. Almost all flags found in the symbol table were the same accross all experiments. This includes setting the register range to 0 to 5, and prioritizing numeric edge cases. We use the flag `select_correct_stack_pointer` during the stack instruction experiment.

6.2.1 Random Instructions We run an experiment where instructions are generated at random. These instructions are generated on three different levels:

1. Random bytes.
2. Random operations, with random values for destination and source register, offset, and immediate values.
3. Random operations, with random values, but values are within accepted ranges.

Random Bytes We run an experiment where the eBPF program, in the `.text` section of the ELF file, is generated using completly random bytes, i.e. no structure is provided by rBPF. To do this, we use the Rust crate *Arbitrary*, used to generate data structures filled with random values, often used in cargo-fuzz fuzzing setups. In this setup, the generator is not run. Instead, we have extended the parser to parse random bytes, and save these in the `.text` section of the ELF file. The generated bytes are completly random, meaning that, as an example, non existing operation codes can be generated, or registers outside 0 to 10.

Our implementation focuses on providing structure to generated eBPF programs, based on the assumption that such programs should be verified as safe to run more often. This experiment is run to gather data on how often valid programs are generated when no structure is provided.

Random Operations with Random Values The second part of this experiment is generating correct operations, but generating random values for the destination and source register, offset, and immediate value. Consider Listing 6, where two instructions are presented. For this experiment, values outside the range of 0 to 10 might be passed to the function `set_dst()`.

Random Operations with Legal Values For the third experiment, values are generated according to eBPF specifications, e.g. registers are generated between values 0 and 10. This is the strategy described in Section 4.4.4 and Section 5.5.2. Considering Listing 6 again, only values between 0 to 10 would be passed to the function `set_dst()`.

Results The results of the experiments are presented in Appendix A. We observe in the results that 0.0% of the programs consisting of random bytes were verified as safe. Only 0.27% of the programs were unmarshaled correctly by PREVAIL, but did not pass verification.

Providing structure through operations resulted in 0.34% of generated programs to be unmarshaled correctly and verified as valid. This further increased with structure through instructions to 0.7% for programs where values are generated within accepted ranges. A shift can also be seen from erroneous programs to invalid programs, as more programs pass the unmarshaling state of PREVAIL, due to correct instruction syntax. This indicates that structured generation facilitates targeting logical error in eBPF program execution.

It can also be observed that from the **Random Operations with Random Values** to **Random Operations with Legal Values** the ratios between erroneous programs and invalid programs greatly shift towards more invalid programs. This happens as programs contain legal values that can then be unmarshaled correctly. Targeting the unmarshaling phase of PREVAIL could thus be done by using the **Random Operations with Random Values** approach.

Random Operations with Random Values was the only experiment where the bug presented in Section 6.3.3 (Invalid Register Use) was found. **Random Operations with Legal Values** was able to find the bugs presented in Section 6.3.2 (Incomplete Load Instruction) and Section 6.3.1 (PREVAIL Segfault).

6.2.2 Stack Instructions We run an experiment where instructions are generated to resemble the use of the program stack, as described in Section 4.4.2 and Section 5.5.2. This experiment consists of two setups:

1. Stack instructions, only using register 10 for load and stores.
2. Stack instructions, using all registers for load and stores.

For both approaches, registers for ALU and jump operations are selected from register 0 to 5.

Stack Instructions Stack Pointer This approach is described in Section 4.4.2 and Section 5.5.2. Programs are generated to utilize the stack frequently, with jump and ALU operations between the structured loads and stores.

Stack Instructions All Registers We change the strategy slightly, such that all registers can be selected as source and destination for load and store instruction. This allows the strategy to utilize context pointers, such as register 1 pointing at network packets.

Results The results of the experiments are presented in Appendix B. The two approaches show very similar results, with a few more programs being verified as safe for the **Stack Instructions All Registers** approach. The results indicate that adding further structure through expert rules and the eBPF specification, increases the ratio of valid programs.

We observe that the **Stack Instructions All Registers** approach results in 0.98% valid programs. This seems counterintuitive as most of the additional registers are not stack or context registers. However, context register 1 is assumed non-null in PREVAIL, and loads and stores to this register are always verified as correct. The additional registers result in an increase in valid programs due to register 1.

The **Stack Instructions All Registers** setup was able to find the bug presented in Section 6.3.5 (uBPF Null Context). During both experiments the strategies were able to find the bug presented in Section 6.3.1 (PREVAIL Segfault).

We observe a difference in how often different bugs occur depending on chosen strategies. The bug presented in Section 6.3.5 is found a few times an hour when deploying the **Random Operations with Legal Values** approach. When we change to the **Stack Instructions All Registers** approach, the bug is encountered multiple times in a single minute. This indicates that strategies can be designed to effectively target certain features and thereby uncover bugs related to this feature.

6.2.3 Rule Break Instructions To test the functionality of the rule break strategy, we allow rule breaks to be generated in previous strategies. We thus have two setups:

1. Rule breaks inserted between random instructions.
2. Rule breaks inserted between stack instructions.

For random instructions we add the rule break strategy to the **Random Operations with Legal Values** setup. For stack instructions we add the rule break strategy to the **Stack Instructions Stack Pointer** setup.

Rule Break Random Instructions We add a chance to generate a rule break when generating random instructions. This includes having long sequences of ALU instructions, in order to break the register counts done by the target verifier.

Rule Break Stack Instructions We add a chance to generate a rule break when generating stack instructions. This includes attempts to write to register 10, i.e. the stack pointer.

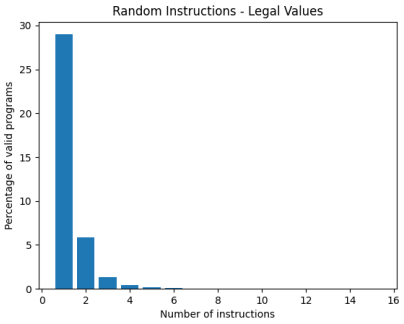
Results The results of the experiments are presented in Appendix C. We observe that the **Rule Break Random Instructions** approach results in more valid programs, compared to the **Rule Break Stack Instructions** approach. This continues the pattern observed in previous experiments, i.e. an increase in valid programs when shifting from random to structured instruction generation.

We observe an increase in erroneous programs, compared to the other experiments. This is likely due to PREVAIL catching any individual instructions that break a rule, when unmarshaling the generated program. This includes instructions attempting to write to register 10, or jump instructions branching out of bounds.

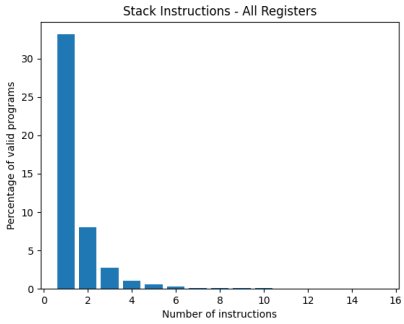
The approach of **Rule Break Random Instructions** was able to find the bug presented in Section 6.3.2. The segmentation fault found in PREVAIL was not found, but it should be possible, as no changes were made affecting the occurrence of this bug.

6.2.4 Experiment Discussion We observe that the strategy based generation approach is beneficial for providing structure to eBPF programs. In Figure 12, we show the change from random instructions to stack instructions. The provided structure results in more programs getting through the unmarshaling state of PREVAIL. Getting through the unmarshaling state means that errors can be found during the verification or execution of eBPF programs.

Experiment results also showed that generating target input based on the chosen strategies allowed our fuzzing harness to target certain eBPF technology features. This is observed by the increased frequency of a given bug related to a certain feature being found by its target strategy.



(a) A graph over the percentage of programs verified as valid for **Random Operations with Legal Values** over program length. Largest program during the experiment is 6 instructions.



(b) A graph over the percentage of programs verified as valid for **Stack Instructions All Registers** over program length. Largest program during the experiment is 10 instructions.

Fig. 12: Percentage of programs verified as valid over program length. A slight increase can be seen when moving from random instructions to more structured instruction generation using the stack strategy.

Map Experiments It would have been ideal to be able to test the map strategies to see how the ratio of valid programs compares. We were unable to do this, as the implementation of maps in user space eBPF technologies is lacking. We presented maps implementation issues in Section 5.5.2.

The strategies involving maps require a lot of instructions to be deterministically set up such that certain behavior is present. The setup for these map strategies facilitate testing map specific eBPF features, such as pointer arithmetic to find specific bugs, i.e. writing out of bounds. Performing experiments with these strategies could give more insight into the trade off between generating unspecific program behavior and targeting certain errors.

6.3 Found Bugs

In this section we cover the bugs found by our fuzzing harness or discovered during its implementation. The bugs and the components of the eBPF technology they impact, will serve as the second part of our evaluation of the strategies. The covered components and their bugs range from the unmarshaling phase and simple syntactical errors to program execution and segmentation faults. The issues that were found, were reported on their respective repositories, to be assessed by their developers.

To reproduce fixed bugs, use the following command on a commit before the fix:

```
git checkout --recurse-submodules <commit ID>
```

For bugs that have yet to be fixed, but might be in the future, `checkout` to a commit before June 16, 2023.

6.3.1 PREVAIL Segmentation Fault A segmentation fault was found in PREVAIL, occuring when PREVAIL computes the remainder in unsigned modulo operations. This computation is done in Crab, during verification of semantic rules. The segmentation fault was generated with both the random instructions and stack sequence strategy.

The segmentation fault occured in cases where the dividend is lower than the divisor, meaning that the dividend is the remainder. An error occured in the cases where the dividend was null. When the null value dividend was compared against the divisor the segmentation fault occured. The problem was fixed by using an `and` operation (`&&`), short circuiting the rest of the check if the dividend is not a finite value.

We reported the bug at [23], and it was fixed by [5]. To reproduce the bug, the branch at [21] should be used.

6.3.2 Incomplete Load Instruction The random instruction strategy often produced eBPF programs that utilize a double word load instruction on an immediate value, i.e. the 128 bit instruction, described in Section 2.4. PREVAIL verifies these as valid programs, while uBPF throws an exception on the

given load instruction. This points to an error in load instruction handling in PREVAIL, as eBPF programs verified by PREVAIL should be safe to run in uBPF.

An example of an eBPF program resulting in this inconsistency:

```
b700 0000 0000 0000    -    mov64 r0, 0x0
5700 0000 756d 3f5d    -    and64 r0, 0x5d3f6d75
bf01 0000 0000 0000    -    mov64 r1, r0
b703 0000 dc5d 3433    -    mov64 r3, 0x33345ddc
1800 0000 3f3b 1268    -    lddw r0, 0x68123b3f68123b3f
1800 0000 3f3b 1268    -
9500 0000 0000 0000    -    exit
```

The last instruction before the exit instruction is the load immediate double word causing the problem. PREVAIL allows this encoding of the 128 bit instruction, but this implementation is incorrect, according to the eBPF documentation [32]. The load instruction should be encoded as:

```
1800 0000 3f3b 1268    -    lddw r0, 0x68123b3f68123b3f
0000 0000 3f3b 1268    -
```

We reported the bug at [39], and it was fixed by [6]. While marshaling the eBPF bytecode, PREVAIL now throws an exception when encountering an incomplete load instruction. To reproduce the bug, the branch at [20] should be used.

6.3.3 Invalid Registers in ALU Operations During the experiment with the **Random Operations with Random Values** strategy test, a bug was found when generating ALU instructions. Specifically for these instructions, PREVAIL appears to verify the use of invalid registers as correct. An example program:

```
mov64 r0, 0x0
mov64 r3, r15
exit
```

In correct programs, register values range from `0` to `a` in the bytecode. In the experiment we allowed values to include `b` to `f` as well, i.e. registers numbered 11 to 15.

We reported the bug at [22]. The bug has, as of writing, not been assessed or fixed.

6.3.4 uBPF Segmentation Fault A segmentation fault was found in uBPF, which occurs during stack preparation for map lookups in the header part of maps strategies. This bug occurs when the map lookup helper function is called and uBPF attempts to access the map attributes of a given eBPF map.

We are not certain were the error originates, but this indicates that when uBPF attempts to allocate an internal map from the ELF file maps section, it does so incorrectly. This results in the attributes of the map either being inaccessible or none existing, resulting in a segmentation fault when accessed.

We reported this bug at [41], but as maps support is not in primary focus this bug has yet to be assessed and fixed.

6.3.5 uBPF Null Context Pointer The programs generated by our fuzzer are not called with any context arguments, i.e. our context argument pointer, register 1, is null. We have found a possible bug in uBPF, where register 1 is assumed to be a context pointer. uBPF reports a load from the null context register as an out of bounds memory load. A similar error is reported by uBPF for store instructions. An example program:

```
mov64 r0, 0x0
arsh64 r0, r5
ldxw r3, [r1+0x1]
mov64 r4, r1
exit
```

In PREVAIL context is always assumed to be non-null. This is because context can not be null, as hooks or events always pass a valid context when triggered.

We have reported this issue [40], but it is not certain what needs to be changed. This will depend on how PREVAIL and uBPF are implemented together in the eBPF-for-Windows system.

6.3.6 Found Bugs Assessment During the fuzzing experiments run for this project we found five bugs. In the following section we will assess the impact of these bugs, and discuss their placement in the target process states. This will serve as the second part of our assessment of our fuzzing harness approach.

The incomplete load bug was assessed and fixed quickly. This bug was found in the unmarshaling phase of PREVAIL, allowing the incorrect syntax for double word load instructions to slip through.

Another bug, the use of invalid registers, was found in the unmarshaling state of PREVAIL. This bug has yet to be assessed by the PREVAIL developers, but we assess this as a bug, as it should not be possible to use invalid registers.

The error found regarding the null context pointer in uBPF is caused by a discrepancy in what is expected of eBPF program context. This bug is yet to be assessed. This could just be an issue of the eBPF-for-Windows project still being in early development and agreement between PREVAIL and uBPF is not complete.

The PREVAIL segmentation fault has been assessed and fixed. The bug was caused by an oversight in the implementation of semantic rule verification. It was possible for the dividend in a modulo operation to be null. The bug was fixed by updating the check of the dividend.

The uBPF segmentation fault is yet to be assessed and fixed. The bug is caused by maps attributes being accessed. This is likely a result of the maps attributes not being accessible, as the allocated map was not loaded correctly into uBPFs internal representation. As this bug involves eBPF maps and uBPF memory, a segmentation fault could be especially severe.

Bugs, or inconsistencies, were found in a wide range of areas in the interaction between PREVAIL and uBPF, which shows that the approach deployed by Buzzy is useful for finding bugs in eBPF technologies. Buzzy is not only able to find simple syntax errors, but is able to find more complex errors as well.

Bug Exploitability The simple bugs might not be exploitable directly. However, the eBPF programs containing these bugs are still verified. Registers and the load operation are commonly used, which means a lot of programs might contain these bugs. However, eBPF developers commonly use BCC or clang, which would not have generated these errors.

The more severe errors could possibly be exploited. One is placed in PREVAIL as it checks the modulo operation, whereas the other is placed in uBPF as it attempts to access an allocated map. The memory related segmentation fault could be especially severe, as it indicates that something goes wrong when the map is being allocated internally in uBPF. While the maps section successfully passes basic section and memory checks in uBPF, the attributes are not accessible. This could indicate that a map might be specified such that when the maps section bytecode is being allocated, something goes wrong in a malicious fashion in uBPF. While uBPF is a virtual machine, if something goes wrong in the right way, e.g. out of bounds access, it would also affect the local machine.

6.4 Design Goals Discussion

During the development of Buzzy we focused on the phase of fuzz data generation. In this section we assess the approach, and its usefulness for fuzzing eBPF technologies. We first present our thoughts on the goals set for each of the six areas of effective fuzzing design, presented in Section 4.1. We then present our thoughts on Buzzy as a whole.

1. **Process States**: We designed Buzzy with the intention of being able to catch logical errors occurring during the execution of the generated programs. Only one such error was encountered when attempting to use maps in uBPF. Buzzy found more bugs previous to this step, i.e. during unmarshaling and verification of the generated program. While error detection should be expanded, a deciding factor could be that relatively few valid large programs are generated.
2. **Code Coverage**: We designed Buzzy to attain code coverage by targeting specific features, i.e. feature coverage. This approach showed promise, as generated strategies were able to target related bugs, i.e. bugs in store and load operations, when focusing on the stack.

3. **Error Detection**: The error detection approach caught five errors. We believe that improving the implemented method would be beneficial. Error detection could be expanded upon in parallel with more specific generator strategies. Specific programs could be set up to target certain errors. This is the goal of the generator strategies involving maps. Expecting a certain value in register 0 could possibly be used for other strategies.
4. **Reproducibility and Documentation**: Saving ELF files when uBPF does not return a value to register 0 proved to be very useful for reproducibility. Programs could easily be rerun with Python scripts, giving a quick overview of the generated bugs from an experiment.
5. **Reusability**: We believe Buzzy to be easily extendable to other eBPF technologies that also utilize ELF files. One caveat of the implementation is how target processes are launched through standard library calls. More sophisticated ways exist to pass input, save target states, and reload with new input, bypassing launching and exiting every iteration.
6. **Resource Constraints**: We designed Buzzy to be easy to deploy and extend for new users. We believe that this design criteria was met, as the strategy based approach, using the skeleton provided by rBPF, provides an easily extendable and deployable harness.

The approach of strategy based eBPF program generation proved to be a valid way to generate programs, both with regard to the bugs uncovered, but also to targeting certain eBPF features.

The valid programs generated by Buzzy are usually small programs. However, our results indicate that it could easily be extended with strategies that target more complex features, resulting in larger valid programs. This could be done by a complete implementation of the maps strategy, or further development of strategies. For these strategies, the symbol table could be expanded, such that more information and context of the generated program is saved.

7 Related Works

In the following sections we present work related to fuzzing eBPF technologies. In Section 7.1, we present a previous project attempting to fuzz eBPF. The program utilized a three-part structure, attempting to initialize an eBPF map, then performing pointer arithmetic, trying to find writing out of bounds errors. In Section 7.2, we present a fuzzer developed to fuzz a fork of rBPF, used by the blockchain company Solana. In Section 7.3, we present Csmith; a tool for generating C programs to fuzz test C compilers.

7.1 eBPF Pointer Arithmetic Fuzzer

In a blogpost [29], Simon Scannell describes a three part structure to fuzz eBPF programs in kernel space. Scannell presents code snippets in the blogpost, but does not provide a code repository. Another project inspired by the blogpost has

created a fuzzer deploying the same strategy developed by Scannell, providing a code repository with a similar setup. [30].

The strategy deployed by these eBPF fuzzers target a single type of bug: writing out of bounds. To do this, the fuzzer initializes an eBPF map, along with a register pointing to the map. This is followed by a set of randomly generated pointer arithmetic instructions to move the location of where the register is pointing. Finally, instructions are generated to write at the pointer location. This strategy attempts to trick the eBPF verifier by messing up register counts and similar arithmetic performed during the verification process. If this succeeds, a store instruction might use a register that points out of bounds for the eBPF map.

The eBPF programs generated by the fuzzers are generated through three phases:

- **Generate header**: Deterministically generates the code for setting up an eBPF map, and returning a register pointing to the map.
- **Generate body**: Randomly generate pointer arithmetic instructions by using two registers. One register initially contains a pointer to the map. The other register is used in the generated instructions, e.g. by first adding an immediate value to the register, followed by an instruction copying the address of one pointer to the other.
- **Generate footer**: Deterministically generates code for writing a value to the location pointed to by a register.

When the eBPF program has been generated it is first verified by the eBPF verifier. Programs determined to be safe by the verifier are then executed. To test if the generated program was able to write out of bounds the generated map is simply inspected. If something is written to the map, the program did not write out of bounds. If the program was verified and executed, i.e. performing a write as the final instruction, but no value is written to the map, the generated eBPF program must have written out of bounds.

7.2 Fuzzing Solana rBPF Fork

In a blogpost [1], Addison Crumb describes an approach taken to fuzz test a forked branch of rBPF. This fork is created by Solana, a company specializing in blockchain technologies.

The fuzzing setup revolves around generating random bytes, and structuring these as eBPF instructions. Generated random bytes are compared against an enumerator that represents the set of valid eBPF operations. If there is a match between the generated bytes, and an eBPF operation, the generated instruction is added to the program. The fuzzer does not check for further rules, such as correct registers, in order to not over-specialize the fuzzer.

The harness is set up to differential fuzz a JIT compiler and interpreter implemented in the rBPF fork. While the JIT compiler might optimize the code, the result, i.e. register 0, should be the same when the programs terminate. The

fuzzer does not target any specific bugs. The fuzz harness was used to find to bugs; a resource exhaustion bug, and a persistent data corruption bug.

7.3 Csmith

Csmith is a tool for generating C programs in order to perform *random differential* testing, i.e. blackbox fuzzing. A program generated by Csmith can be passed to different compilers, and by comparing the compiled results, it can be determined whether an error exists in one of the compilers. When using three or more compilers, it can be determined heuristically when one produces an error. Csmith has two main design goals:

1. Generated programs must be well formed and adhere to the C standard.
2. Csmith must maximize expressiveness, i.e. utilize as many different C language features as possible. This goal is based on the hypothesis that expressive programs are more likely to find bugs in the compiler.

To generate valid C programs Csmith uses a *global environment* that keeps track of defined variables and call chains, s.t. changes to variables in the generated programs are kept track of. Csmith also uses a grammar representing a subset of the C language to generate program structures.

When Csmith starts generating a program, it firsts generates a set of struct types, filling them with random members of different types. When the struct types have been generated, Csmith starts generating functions starting from a main function. Generating a new piece of C code is done through six sub-steps:

1. Choose an allowed production from the grammar. To determine what variables to use a probability table is used. The table contains variables that can be accessed in the current scope. A filter is then applied, checking for things like maximum statement depth.
2. If a target is needed for the generated production, Csmith chooses an existing one, or declares a new target variable.
3. If a type can be selected for the generated production, Csmith randomly selects one by consulting the global environment, probability table, and filters.
4. If the generated production is nonterminal, Csmith recurses and continues to generate productions until the compound statement is terminated.
5. Csmith then executes a set of dataflow transfer functions. Csmith uses these to update the local environment.
6. Csmith finally performs a set of safety checks. If the generated fragment does not pass the checks, it is not committed to the generated program.

8 Conclusion

We designed and implemented Buzzy, an unguided smart-strategy generation-based blackbox fuzzer for eBPF technologies. When developing Buzzy, we wanted to answer whether a strategy based approach of input generation is useful for this novel tool. We focused primarily on feature coverage, in order to determine if certain features could be targeted, and if bugs or other interesting behavior of the chosen targets could be uncovered.

We conclude that the developed strategies proved to be useful for input generation, both considering feature coverage, bug discovery and bug targeting. We base this on an increase in valid programs and certain bugs being found more often, when applying strategies targeting a given feature.

Buzzy is easily applied to other targets, and bugs are easily replicated. Buzzy is currently limited to targeting simple features and early process states, as few large valid programs are generated. We do believe that these limitations can be solved. We therefore conclude that Buzzy sufficiently addresses the six requirements for efficient fuzzing, but with room to improve.

9 Future Work

The approach of our fuzzing harness design and the developed strategies showed that strategies can be used to target certain eBPF features. As strategies providing more structure to the eBPF program was deployed, an increase was seen in valid programs. We could also observe, that certain strategies were able to uncover bugs related to the strategy more often, than general purpose strategies. In this section we will cover how Buzzy could be developed further upon by adding to, changing, or enhancing the different fuzzing harness components. We consider changes that could result in a better fuzzing harness while still reflecting the design goals established for Buzzy.

Fault Detection We could extend our fault detection method, such that we are able to extract and utilize more of the information available after execution. Currently, errors are logged when there is a discrepancy between what the two targets, PREVAIL and uBPF, determine as correct. Additionally a fault detection method was developed for the eBPF maps strategies that considers the maps memory discrepancy. Assuming these eBPF technologies continue developing the maps and memory features, both of these paths could be improved upon.

Execution output sometimes provide details regarding the specific instruction that causes a given error. This could be used to provide some preliminary classification, possibly eliminating part of the manual process during bug analysis.

We could design a strategy that further targets the maps and memory features of the eBPF technology by performing a sequence of instructions with the aim of knowing the resulting value. If the outcome of a memory based strategy

is known, the memory discrepancy detection technique could be extended, such that it checks for a given value returned after program termination. This could facilitate detection of faulty value arithmetic and testing of memory allocation features, as we expect a certain value to be found in memory.

Taming During generation, a lot of similar eBPF programs are produced. This results in some of the errors found by fault detection being logged multiple times across executions. We could extend the fault detection process such that it also keeps track of the kind of errors it is logging. Introducing awareness of previously logged errors, could help avoid logging the same errors. This is called taming [4].

Another simple technique, would be minimizing the set of logged ELF files afterwards. This would require each error log file to be rerun, which is not as efficient as the first method. However, minimizing afterwards could result in some extra control over certain aspects, such as finding the smallest and largest example of a given error.

Program Type Context Aware Generation The design of the strategies deployed by Buzzy, focuses mainly on covering the different features of the eBPF technology and their structures. To extend the coverage of features, additional strategies could be derived from the eBPF specification with regards to the context of each type of eBPF program.

Designing for context awareness requires deriving specifications for each eBPF program type. Each strategy based on the type dependant aspects of an eBPF program, would target a fairly narrow subset of features. Deriving these strategies from the eBPF specification requires a fair amount of manual work, but could result in improved feature coverage.

Most available eBPF program samples are specific to a given technology that builds upon eBPF, utilizing their given contexts fairly complexly. Context aware eBPF program generation with a mutation-based approach, could utilize these eBPF samples, to more easily facilitate context awareness.

Guided Generation As covered in our evaluation, we observe that a lot of the eBPF that were valid contained very few instructions. Our fuzzing technique could be changed such that, instead of generating a new eBPF program during each iteration, one instruction could be generated and appended at a time. This technique would verify increasingly large eBPF programs. The logic behind this technique is the assumption that, the more instructions that are executed the more complex behavior the eBPF program is able to perform.

Move away from LibFuzzer One component of our fuzzing harness is the tool used for fuzzing in Rust. The cargo-fuzz tool is able to facilitate coverage guided fuzzing through LibFuzzer when deployed against a target written in Rust, but this feature was not used, as our targets are written in C/C++. The way our fuzzing harness is setup to fuzz our targets, LibFuzzer is only used as a

fuzzing iteration engine. This means that LibFuzzer, as we use it, is disconnected from the coverage based feedback mechanisms. One aspect of LibFuzzer that has limited use, is the random bytes generated as input for the fuzzing targets. This is only used for the random bytes for the random bytes instructions strategy and the random maps strategy.

For most of our strategies we simply use a random integer generator provided by standard library in Rust. The randomly generated integers are used for the distribution of random instructions during the generation loops.

This process could be implemented as part of Buzzy's own fuzzing iteration engine, by simply looping for the required amount of instructions and utilizing the same random integer generator through the standard library. This would also enable greater control of information across fuzzing iterations.

A Random Instructions Experiment Results

Results for the experiments presented in Section 6.2.1.

-	Count	Percentage
Total number of programs	858.098	-
Valid programs	0	0.0%
Invalid programs	2.326	0,27%
Erroneous programs	671.349	78,24%
Parsing failed	184.423	21,49%

Table 3: Results after running our fuzzing harness for 1 hour for the **Random Bytes** experiment setup.

-	Count	Percentage
Total number of programs	1.486.030	-
Valid programs	5.010	0,34%
Invalid programs	37.372	2,51%
Erroneous programs	1.339.823	90,16%
Parsing failed	103.825	6,99%

Table 4: Results after running our fuzzing harness for 1 hour for the **Random Operation with Illegal Values** experiment setup.

-	Count	Percentage
Total number of programs	996.751	-
Valid programs	7.056	0,7%
Invalid programs	378.059	37,92%
Erroneous programs	450.726	45,21%
Parsing failed	160.910	16,14%

Table 5: Results after running our fuzzing harness for 1 hour for the **Random Operation with Legal Values** experiment setup.

B Stack Sequence Experiment Results

Results for the experiments presented in Section 6.2.2.

-	Count	Percentage
Total number of programs	867.494	-
Valid programs	6.987	0,81%
Invalid programs	302.522	34,87%
Erroneous programs	509.857	58,77%
Parsing failed	48.128	5,55%

Table 6: Results after running our fuzzing harness for 1 hour for the **Stack Sequence Only Register 10** experiment setup.

-	Count	Percentage
Total number of programs	891.655	-
Valid programs	8.711	0,98%
Invalid programs	300.847	33,74%
Erroneous programs	510.299	57,23%
Parsing failed	71.798	8,05%

Table 7: Results after running our fuzzing harness for 1 hour for the **Stack Sequence All Registers** experiment setup.

C Rule Break Experiment Results

Results for the experiments presented in Section 6.2.3.

-	Count	Percentage
Total number of programs	892.421	-
Valid programs	1.464	0,16%
Invalid programs	144.114	16,16%
Erroneous programs	677.910	75,96%
Parsing failed	68.933	7,72%

Table 8: Results after running our fuzzing harness for 1 hour for the **Rule Break in Random Instructions** experiment setup.

-	Count	Percentage
Total number of programs	745.921	-
Valid programs	2.545	0,34%
Invalid programs	146.115	19,59%
Erroneous programs	556.405	74,59%
Parsing failed	40.856	5,48%

Table 9: Results after running our fuzzing harness for 1 hour for the **Rule Break in Stack Sequence** experiment setup.

References

1. Addison Crump: Earn $200K by fuzzing for a weekend: Part 1. https://secret.club/2022/05/11/fuzzing-solana.html, accessed: June 15, 2023
2. Bauman, Laura: Harnessing the eBPF Verifier. https://blog.trailofbits.com/2023/01/19/ebpf-verifier-harness/, accessed: June 15, 2023
3. Bounimova, E., Godefroid, P., Molnar, D.: Billions and billions of constraints: Whitebox fuzz testing in production. In: 2013 35th International Conference on Software Engineering (ICSE). pp. 122–131 (2013). https://doi.org/10.1109/ICSE.2013.6606558
4. Chen, Y., Groce, A., Zhang, C., Wong, W.K., Fern, X., Eide, E., Regehr, J.: Taming compiler fuzzers. In: Proceedings of the 34th ACM SIGPLAN conference on Programming language design and implementation. pp. 197–208 (2013)
5. Dave Thaler: Fix segfault in URem. https://github.com/vbpf/ebpf-verifier/pull/500, accessed: June 15, 2023
6. Dave Thaler: Reject invalid LDDW instructions. https://github.com/vbpf/ebpf-verifier/pull/486, accessed: June 15, 2023
7. eBPF Foundation: eBPF Documentation. https://ebpf.io/what-is-ebpf/, accessed: June 15, 2023
8. Geretto, E., Giuffrida, C., Bos, H., Van Der Kouwe, E.: Snappy: Efficient fuzzing with adaptive and mutable snapshots. In: Proceedings of the 38th Annual Computer Security Applications Conference. pp. 375–387 (2022)
9. Godefroid, P., Levin, M.Y., Molnar, D.: Automated whitebox fuzz testing (November 2008)
10. Google: Google Kubernetes Engine Documentation. https://cloud.google.com/kubernetes-engine/docs, accessed: June 15, 2023
11. Google: New GKE Dataplane V2 increases security and visibility for containers. https://cloud.google.com/blog/products/containers-kubernetes/bringing-ebpf-and-cilium-to-google-kubernetes-engine, accessed: June 15, 2023
12. International Organization for Standardization: ISO/IEC 9899:TC2. https://open-std.org/jtc1/sc22/wg14/www/docs/n1124.pdf, accessed: June 15, 2023
13. IO Visor Project: BPF Compiler Collection (BCC). https://github.com/iovisor/bcc, accessed: June 15, 2023
14. IO Visor Project: Userspace eBPF VM. https://github.com/iovisor/ubpf/, accessed: June 15, 2023
15. Liang, J., Wang, M., Zhou, C., Wu, Z., Jiang, Y., Liu, J., Liu, Z., Sun, J.: Pata: Fuzzing with path aware taint analysis. In: 2022 IEEE Symposium on Security and Privacy (SP). pp. 1–17 (2022). https://doi.org/10.1109/SP46214.2022.9833594
16. Linux: bpf - Linux manual page. https://www.man7.org/linux/man-pages/man2/bpf.2.html, accessed: June 15, 2023
17. LLVM Project: libFuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, accessed: June 15, 2023
18. Michał Zalewski: Binary fuzzing strategies: what works, what doesn't. https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html, accessed: June 15, 2023
19. Microsoft: eBPF for Windows. https://github.com/microsoft/ebpf-for-windows, accessed: June 15, 2023
20. Microsoft: Incomplete load instruction reproduction branch. https://github.com/vbpf/ebpf-verifier/tree/57d1aa78a5cff65f8d49aeba4778759c73bf60ce, accessed: June 15, 2023

21. Microsoft: PREVAIL segmentation fault reproduction branch. https://github.com/vbpf/ebpf-verifier/tree/47ff28219a4909cb0a26b75d164100926bffd014, accessed: June 15, 2023
22. Mikkel Tolstrup Jensen: PREVAIL appears to verify use of invalid registers as correct for ALU operations. https://github.com/vbpf/ebpf-verifier/issues/505, accessed: June 15, 2023
23. Mikkel Tolstrup Jensen: Segmentation fault (core dumped). https://github.com/vbpf/ebpf-verifier/issues/493, accessed: June 15, 2023
24. Monnet, Quentin: Rust virtual machine and JIT compiler for eBPF programs. https://github.com/qmonnet/rbpf, accessed: June 15, 2023
25. Nilizadeh, S., Noller, Y., Pasareanu, C.S.: Diffuzz: Differential fuzzing for side-channel analysis. In: 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE). pp. 176–187 (2019). https://doi.org/10.1109/ICSE.2019.00034
26. Pham, V.T., Böhme, M., Roychoudhury, A.: Aflnet: a greybox fuzzer for network protocols. In: 2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST). pp. 460–465. IEEE (2020)
27. Rice, L.: Learning eBPF. O'Reilly Media, Inc. (2023)
28. Rust Fuzzing Authority: cargo fuzz. https://github.com/rust-fuzz/cargo-fuzz, accessed: June 15, 2023
29. Simon Scannell: Fuzzing for eBPF JIT bugs in the Linux kernel. https://scannell.io/posts/ebpf-fuzzing/, accessed: June 15, 2023
30. Snorez: Snores eBPF Fuzzer. https://github.com/snorez/ebpf-fuzzer, accessed: June 15, 2023
31. Sutton, M., Greene, A., Amini, P.: Fuzzing: Brute force vulnerability discovery (01 2007)
32. Thaler, D.: eBPF Instruction Set Specification, v1.0. Internet-Draft draft-thaler-bpf-isa-00, Internet Engineering Task Force (Mar 2023), https://datatracker.ietf.org/doc/draft-thaler-bpf-isa/00/, work in Progress
33. The Cilium Authors: eBPF-based Networking, Observability, Security. https://cilium.io/, accessed: June 15, 2023
34. The Falco Authors: The Falco Project. https://falco.org/, accessed: June 15, 2023
35. The kernel development community: 1 eBPF Instruction Set Specification, v1.0. https://docs.kernel.org/bpf/instruction-set.html, accessed: June 15, 2023
36. The kernel development community: eBPF verifier. https://docs.kernel.org/bpf/verifier.html, accessed: June 15, 2023
37. The Rust Team: Install Rust. https://www.rust-lang.org/tools/install, accessed: June 15, 2023
38. The vBPF Organization: PREVAIL - a Polynomial-Runtime EBPF Verifier using an Abstract Interpretation Layer. https://github.com/vbpf/ebpf-verifier, accessed: June 15, 2023
39. Tobias Bruun Sandberg Hansen: Inconsistency between PREVAIL verifier and uBPF. https://github.com/vbpf/ebpf-verifier/issues/484, accessed: June 15, 2023
40. Tobias Bruun Sandberg Hansen: Out of bounds memory load, store - PREVAIL assumes context is non-null but uBPF sets context to null. https://github.com/iovisor/ubpf/issues/279, accessed: June 15, 2023
41. Tobias Bruun Sandberg Hansen: Segmentation fault caused by call instruction. https://github.com/iovisor/ubpf/issues/247, accessed: June 15, 2023

42. Zeller, A., Gopinath, R., Böhme, M., Fraser, G., Holler, C.: The Fuzzing Book. CISPA Helmholtz Center for Information Security (2023), https://www.fuzzingbook.org/, retrieved 2023-01-07 14:37:57+01:00