Understanding the Security of Linux eBPF Subsystem

Mohamed Husain Noor Mohamed
Virginia Tech
Blacksburg, USA
nkhusain@vt.edu

Xiaoguang Wang*
University of Illinois Chicago
Chicago, USA
xgwang9@uic.edu

Binoy Ravindran
Virginia Tech
Blacksburg, USA
binoy@vt.edu

ABSTRACT

Linux eBPF allows a userspace application to execute code inside the Linux kernel without modifying the kernel code or inserting a kernel module. An in-kernel eBPF verifier pre-verifies any untrusted eBPF bytecode before running it in kernel context. Currently, users trust the verifier to block malicious bytecode from being executed.

This paper studied the potential security issues from existing eBPF-related CVEs. Next, we present a generation-based eBPF fuzzer that generates syntactically and semantically valid eBPF programs to find bugs in the verifier component of the Linux kernel eBPF subsystem. The fuzzer extends the Linux Kernel Library (LKL) project to run multiple lightweight Linux instances simultaneously, with inputs from the automatically generated eBPF instruction sequences. Using this fuzzer, we can outperform the `bpf-fuzzer` [10] from the `iovisor` GitHub repository regarding fuzzing speed and the success rate of passing the eBPF verifier (valid generated code). We also found two existing ALU range-tracking bugs that appeared in an older Linux kernel (v5.10).

CCS CONCEPTS

• **Security and privacy** → **Operating systems security**; *Domain-specific security and privacy architectures.*

KEYWORDS

Linux eBPF, Kernel Security, Fuzzing

ACM Reference Format:
Mohamed Husain Noor Mohamed, Xiaoguang Wang, and Binoy Ravindran. 2023. Understanding the Security of Linux eBPF Subsystem. In *14th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '23), August 24–25, 2023, Seoul, Republic of Korea.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3609510.3609822

*X. Wang's work was done while he was at Virginia Tech.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for third-party components of this work must be honored. For all other uses, contact the owner/author(s).

APSys '23, August 24–25, 2023, Seoul, Republic of Korea
© 2023 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-0305-8/23/08.
https://doi.org/10.1145/3609510.3609822

1 INTRODUCTION

The eBPF (extended Berkeley Packet Filter) has emerged as one of the most exciting techniques in the Linux kernel. Introduced in Linux kernel v3.18, eBPF has since gained widespread popularity for enabling kernel state observability and network and system behavior control. It has become essential to modern network monitoring systems and control infrastructure [14, 23].

eBPF allows programmers to safely execute Berkeley Packet Filter (BPF) bytecode within the Linux kernel without inserting a kernel module or modifying the kernel source code [23]. The eBPF kernel subsystem provides a system call to load the userspace bytecode into the kernel and attach them to eBPF probes [19]. Such probes are predefined hook points on many kernel code paths, including network events, system calls, function entry and exit, and kernel tracepoints [22].

To prevent malicious or buggy eBPF programs from crashing the kernel or causing other security issues, an in-kernel eBPF verifier performs a series of static checks on the eBPF bytecode to ensure that it adheres to a set of safety rules, such as pointer boundary safety, type safety, and no endless loops [13]. The eBPF verifier plays a critical role in ensuring the safety and security of the eBPF infrastructure within the Linux kernel. Though the verifier is a security-critical component of the eBPF runtime, like any software, it contains vulnerabilities that could potentially affect the security of the entire kernel. With these vulnerabilities, a program verified as valid by the verifier may still compromise the kernel, leading to privilege escalation and denial of service attacks (*e.g.,* CVE-2016-4557 [3], CVE-2021-3490 [4]).

This paper systematically analyzes the potential attack surface for the in-kernel eBPF subsystem. We first looked at existing eBPF-related bugs and summarized them based on their root causes. Next, we report a work-in-progress lightweight eBPF fuzzer to identify security vulnerabilities in the Linux kernel eBPF verifier component. The fuzzer automatically generates valid eBPF instructions to observe the misbehavior of the verifier. Moreover, our fuzzer completely

runs in userspace and can scale out the fuzzing process with hundreds of threads on a single server machine.

With this fuzzer, we were able to identify two ALU range-tracking bugs previously reported in the Linux kernel. We instrumented the verifier, reaching 40 percent code coverage which covers almost all of the functions related to the verifier's ALU-related code. With 32 parallel threads, the fuzzer can generate and execute 300 eBPF programs per minute which can stress the correctness of the verifier.

The rest of the paper is organized as follows: Section 2 provides background information on the eBPF subsystem. Section 3 describes the design and implementation of the eBPF fuzzer. The evaluation is presented in Section 4. We discuss possible optimizations in Section 5. We contrast related work in Section 6 and conclude the paper in Section 7.

2 BACKGROUND AND MOTIVATION

2.1 Linux eBPF and Security

The Berkeley Packet Filter, or classic BPF, is a technology available in specific operating systems to analyze network traffic. BPF allows the userspace program to inject a filter program that specifies the conditions to receive a packet. This dramatically improves the performance as the kernel drops the unwanted packets without forwarding them to the userspace level. The recent Linux kernel introduced an extended Berkeley Packet Filter (eBPF). In addition to network traffic filtering, it can be utilized for performance monitoring, security enhancement, and debugging.

The eBPF programs can be written in high-level languages (like C or Rust) or eBPF assembly format. Once the eBPF program is compiled into eBPF bytecode, the `bpf()` system call can be used to load the corresponding eBPF bytecode into the Linux kernel [19]. An eBPF verifier verifies the correctness before the kernel executes the program on selected events. Linux eBPF provides ten 64-bit general-purpose registers and a read-only frame pointer. The instruction set includes load and store instructions and support for arithmetic operations like addition, subtraction, multiplication, and bit operations (e.g., `AND`, `OR`, and `XOR`). Another instruction class is the conditional and unconditional `JMP` instructions for changing the program's control flow.

As mentioned earlier, eBPF program execution is event-driven. An event can be a system call, function entry, exit, perf event, kprobes, and uprobes. The input context to the eBPF program varies based on the hook/event it is attached. For example, when an eBPF program is attached to a system call, the arguments of the system call will be passed as the context. Similarly, for a kernel tracepoint, the data associated with the tracepoint is passed as an input. Although an eBPF program needs to be verified before being executed, exploits are still reported that leverage vulnerabilities in the eBPF subsystem to privilege escalation [17].

To understand security vulnerabilities in the Linux eBPF subsystem, we manually examined Linux eBPF-related CVEs in the past two years [5]. The vulnerabilities reported on eBPF are categorized in Figure 1. Looking at existing eBPF-related CVEs, we found that almost half of the vulnerabilities reported are related to the eBPF verifier. The eBPF helper has the second-highest number of the CVEs reported. However, we found several bugs in other eBPF components can be fixed within the eBPF verifier.

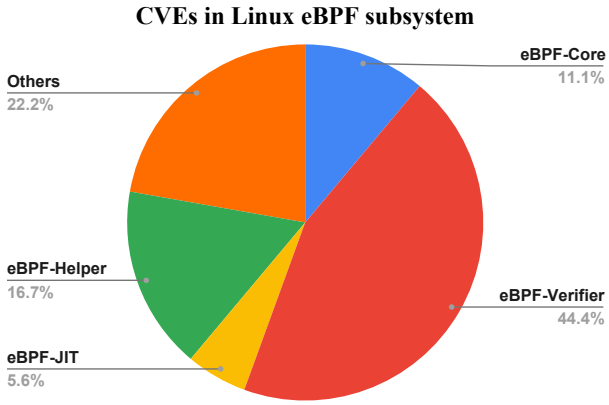

Figure 1: Categories of vulnerabilities in Linux eBPF subsystem.

We further investigated the characteristic of bugs in the eBPF verifier and found that four of the eight vulnerabilities are related to the ALU range tracking operations. The remaining CVEs are related to integer overflow and improper input validation (some are listed in Table 1). The ALU range tracking bugs are a class of bugs where the verifier incorrectly computes the possible range of the registers for each instruction in the program. This will lead to the improper assumption that certain pointer arithmetics are valid and that the resulting memory access is inbound. For example, the verifier with the vulnerabilities may assume the register `r2` value as 10, but the actual runtime value will be 25 or any value other than 10. If the `r2` register is used for pointer arithmetic for cases like accessing a member of the map, the access may go beyond the actual size of the map, which the verifier is supposed to detect. But because of the ALU range tracking vulnerabilities, the attacker could exploit this to steal or modify sensitive information.

2.2 Kernel Fuzzing

Kernel fuzzers have been proven as a promising technique for identifying software vulnerabilities [9]. The operating

Table 1: CVEs related to eBPF verifier.

CVE ID	Description	Type
CVE-2021-4204	OOB in bpf_ringbuf_submit	Improper Input Validation
CVE-2022-0264	check_mem_access :Internal memory locations returned to userspace	Kernel Address Leakage
CVE-2022-23222	Pointer arithmetic via *_OR_NULL pointer types	ALU Range Tracking error
CVE-2020-8835	Invalid range in __reg_bound_offset32	ALU Range Tracking error
CVE-2021-3490	ALU-32 bounds tracking for bitwise ops - AND, OR and XOR	ALU Range Tracking error
CVE-2020-27194	scalar32_min_max_or mishandles bounds tracking during use of 64-bit values	ALU Range Tracking error

system kernel bugs allow attackers to access a system with full privileges. Fuzzing generally involves generating random inputs to the target program or kernel and observing the behavior of the systems with unconventional inputs. In our proposed eBPF fuzzer, the system under test is the verifier component of the eBPF run-time, and the input is an eBPF bytecode.

Though well-established user-space fuzzers like AFL [26] and AFL++ [8] are already available, they are unsuitable for fuzzing the eBPF verifier. The verifier expects the eBPF instructions to be syntactically and semantically valid. The conventional fuzzers modify certain bits of the inputs for each cycle and observe the program's behavior through a feedback mechanism like code coverage. Thus Fuzzer needs to be aware of the eBPF instruction set and semantics to generate the inputs that can pass the verifier.

3 A LIGHTWEIGHT LINUX EBPF FUZZER

We designed and implemented a lightweight Linux eBPF fuzzer to identify security flaws in the Linux kernel eBPF verifier (Figure 2). The key idea is to generate eBPF bytecode programs that lead to logical violations in the eBPF verifier. The fuzzer has two major components. The first is a fuzzer manager responsible for launching multiple kernel-eBPF instances that load and execute randomly generated eBPF programs. The other major component is the eBPF program generator that produces the random eBPF instructions, which are semantically and syntactically valid, so the verifier does not directly drop the test program.

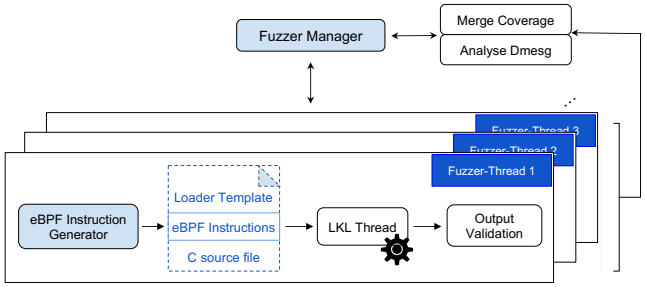

Figure 2: System overview of the lightweight Linux eBPF Fuzzer.

3.1 Generate Random eBPF Instructions

The eBPF instruction set is similar to a RISC register machine. The eBPF instruction set comprises eight classes of instructions, mainly in three categories – ALU, LOAD/STORE, and (conditional) JMP. Each eBPF instruction has 64 bits in size. For each instruction, the lower 8 bits are encoded for the opcode (Figure 3). The opcode's three least significant bits (LSB) encode the instruction class. The destination register and source register are encoded right after the opcode field. Currently, eBPF supports up to 11 registers (`r0-r10`) to be used for eBPF programs.

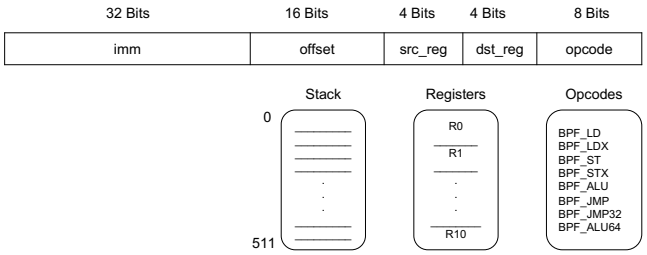

Figure 3: eBPF instruction encoding [12].

For each fuzzing cycle, the fuzzer generates a random number (N) of instructions and randomly chooses one instruction class for each instruction. The generated eBPF instructions will then be inputted into the Linux eBPF instances to test whether they can trigger a bug for the verifier. One main challenge for the fuzzer is to generate random instructions that can pass the verifier but are random enough to trigger bugs in the verifier. To solve this problem, the fuzzer creates syntactically valid instructions by enforcing the grammar of the eBPF instruction set [12]. Each instruction generator follows the rules for its instruction class so that randomly generated instruction passes the compiler and eBPF verifier without any error.

Another challenge is to generate a set of instructions that is also semantically valid. For instance, an eBPF ALU instruction that does a division operation with 0 as a divisor has correct syntax and can compile without failure. But the verifier would throw an error, as it is supposed to ensure the program's safety, mark it as an invalid program, and drop the program before attaching it to a kernel event. Another

example could be a load instruction that copies a value from an uninitialized register to a destination register. This is an invalid operation since loading from an uninitialized register would lead to undefined behavior.

To solve this problem, our fuzzer randomly allocates registers for each instruction and generates the register values. However, the fuzzer's instruction generator scans the program for uninitialized registers and initializes those registers after finishing the instructions generation. The instruction generator omits the instructions that can are considered invalid. For example, shift operations like `BPF_RSH` and `BPF_LSH` cannot have negative shift values. The `BPF_JMP` and `xBPF_JMP32` instructions cannot have negative lengths (i.e., back edge), and the destination should not go beyond the last instruction. Another key aspect of the instruction generator is the selection of register values. The registers can have 32-bit and 64-bit values. We found that some immediate values are more accessible to trigger ALU bugs, such as `0x0`, `0x1`, `0x2`, `0x4`, `0x8`, `0xf`, `0xff`, `0xffff`, `0xffffffff`. Therefore, we prioritize these values as immediate values for instructions and assign them to registers.

Lastly, suppose one randomly generated instructions sequence passes the eBPF program verification and increases the code coverage for the eBPF subsystem. In that case, we prioritize such a program into a seed pool and mutate register values for the next fuzzing cycle. The generated eBPF instructions are stored as a seed program and then converted to a C program with macros for each eBPF instruction.

3.2 Efficiently Fuzzing the eBPF Verifier

Once eBPF programs are generated, we input them to live kernel instances. Our fuzzer aims to launch many lightweight Linux kernel instances to run selected randomly generated eBPF programs. Since we target finding bugs for the actual Linux eBPF subsystem, one strategy is to launch Linux virtual machines (VMs) and execute eBPF programs within the VM. However, the cost of starting/stopping VMs is high. Instead, we extended a lightweight and userspace Linux – Linux Kernel Library (LKL) [15] – to execute randomly generated eBPF programs (Figure 4).

The LKL project allows users to compile the kernel code into an object file (i.e., a library) so that applications can directly link with the kernel code. LKL is implemented as an architecture port in the `arch/lkl` folder of the Linux source. It can be used to execute the kernel code in the userspace. Our eBPF fuzzer launches the `bpf()` syscall within LKL instances, thus invoking the eBPF verifier as a userspace application. This allows multiple verifier instances to be invoked as separate threads, thus increasing the throughput of the fuzzer. As opposed to fuzzing userspace programs, one of the main limitations of fuzzing kernel components is that, at a time, only one set of inputs can be executed in a single system or a virtual machine. For example, syzkaller, the state-of-the-art kernel fuzzer, invokes virtual machines instance to test multiple kernels in parallel [9]. This requires a lot of computing resources, even for testing a small part of the kernel.

```
➜   [fuzzer] python3 ebpf_gen.py
Time:1  Pass:0 Fail:0 Total:0 Speed:0.0 AE=0
Time:2  Pass:0 Fail:0 Total:0 Speed:0.0 AE=0
Time:3  Pass:7 Fail:0 Total:7 Speed:2.1 AE=0
Time:4  Pass:14 Fail:0 Total:14 Speed:3.2 AE=0
Time:5  Pass:22 Fail:0 Total:22 Speed:4.1 AE=0
Time:6  Pass:22 Fail:0 Total:22 Speed:3.5 AE=0
Time:7  Pass:23 Fail:0 Total:23 Speed:3.1 AE=0
Time:8  Pass:31 Fail:0 Total:31 Speed:3.7 AE=0
Time:9  Pass:35 Fail:0 Total:35 Speed:3.7 AE=0
Time:10  Pass:43 Fail:0 Total:43 Speed:4.1 AE=0
```

Figure 4: Screenshot of the eBPF fuzzer execution.

Our eBPF fuzzer is enclosed in a loader program which includes the code for calling `lkl_syscall(BPF_PROG_LOAD)`. Next, the fuzzer manager binds our extended LKL instance with a random eBPF program and runs it as an LKL thread. Each LKL thread boots the kernel as a userspace application and loads the generated eBPF program. The fuzzer also triggers a socket event that, in turn, executes the verified eBPF program. If the verifier passes it, a socket event is triggered so that the kernel executes the loaded eBPF program. After the execution, the fuzzer verifies the kernel logs and checks if errors are reported.

Another benefit of testing the Linux eBPF subsystem in userspace is we can easily apply various userspace sanitizers to the target. For example, we can compile the LKL eBPF subsystem with LLVM address sanitizer to enable memory safety checks [16]. Moreover, we insert assertions for the eBPF ALU logic code. For example, we add conditions to catch an invalid register state. Specifically, we store all registers' legal states (i.e., value ranges) right after the eBPF verification phase. We use the register states calculated from the verifier to validate the actual register state. This allows us to verify whether actual register values are maliciously manipulated and exploited by the eBPF runtime (e.g., a miscalculated case by the verifier).

4 EVALUATION

We ran our early prototype of the LKL-based eBPF fuzzer on a machine with an Intel Xeon D-1548 CPU of 16 cores and 62 GB of memory. The fuzzer launches 32 parallel LKL Linux instances and runs 300 eBPF programs per minute. We tested our fuzzer in Linux kernel version 5.10.0.

Figure 5 shows the code coverage of the eBPF verifier increases according to the time executed. After 30 minutes of execution, our eBPF fuzzer can cover 40% of the verifier

code and almost all ALU-related functions. Among the remaining unreached branches is code for BPF type format (BTF) [11], BPF maps, and memory access validation functions. They hold 10 percent each. The remaining 30% of unreached branches are in pointer and argument validation. This is due to the lack of additional checks in those code regions in our current eBPF fuzzer.

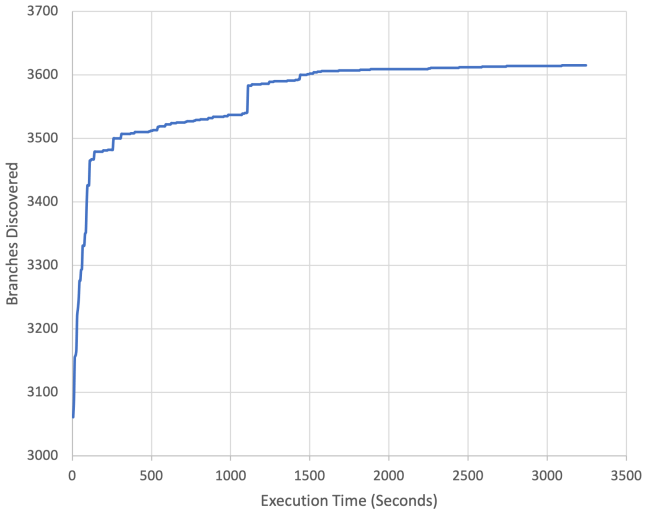

Figure 5: Code coverage numbers are discovered in the eBPF verifier (9730 branches in total) according to the execution timeline.

Our current design allows more than 50 percent of automatically generated eBPF programs to pass the verifier without rejection. Compared to other open-source eBPF fuzzers [10, 18], our fuzzer has much higher fuzzing throughput. For example, the eBPF fuzzer from the `iovisor` project [10] can only generate 4 *valid eBPF programs* out of 2000 generated eBPF programs, rendering a high percentage of generated eBPF programs fail to execute directly. The existing implementations lack generating valid programs that can pass the verifier, in contrast to our grammar-aware eBPF program generator. Moreover, our lightweight fuzzer implementation allows multiple eBPF programs to be executed in parallel.

Our eBPF fuzzer can trigger two existing ALU range-related bugs within the first 5 minutes. In particular, one eBPF verifier vulnerability (`CVE-2022-23222` [7]) is related to `OR_NULL` pointer type, where pointer arithmetic is allowed on `NULL` pointers. `OR_NULL` pointer is an intermediate pointer type where a pointer could have a NULL value. In the runtime, the pointer is checked to determine whether it is NULL. The vulnerability allows an attacker to execute arithmetic operations on the NULL pointer when the verifier sees it is a valid operation otherwise. So the attacker can do arbitrary read/write on any out-of-bound memory. The second vulnerability we found (`CVE-2020-27194` [6]) is a result of improper range tracking in the verifier that could be used to run Out-of-Bound memory access. During the verification process, the eBPF verifier tracks the possible range of the register values and their states to identify OOB memory access. The `scalar32_min_max_or()` function in the verifier used the 64-bit registers instead of 32-bit to calculate the minimum and maximum values.

5 DISCUSSION

Our eBPF fuzzer has some limitations and improvement spaces. The current implementation concentrates only on the ALU-related issues in the verifier. There are other sections in the verifier where this fuzzer can be extended to test those sections. For example, the verifier must validate instructions interacting with *eBPF helper functions* and *map data structures* before executing them in the kernel. The eBPF generator can be modified to generate those instructions with proper grammar, which will also increase the code coverage in the verifier.

Our eBPF fuzzer is built on top of the Linux Kernel library to execute eBPF programs. However, the LKL project is a little outdated to the most recent Linux kernel. Therefore, a few most recent eBPF helper functions do not work well with even the latest LKL. Moreover, there are some eBPF helper functions that LKL does not support well due to LKL's userspace property. Hence to extend the fuzzer for eBPF helper functions, it is necessary to add support for these helper functions in LKL. We leave it as future work.

The eBPF fuzzer can also be enhanced with additional sanitizers that would identify dynamic memory errors in the verifier. Address/memory sanitizers [16, 20] are commonly used in kernel fuzzers, but they would not catch the issues related to the correctness of the verifier. Nevertheless, these sanitizers could identify potential memory safety vulnerabilities in the verifier.

6 RELATED WORK

Recently, kernel fuzzing has become a hot topic among security researchers. Some fuzzers test the entire kernel through system call interface [21] inputs, and other fuzzers concentrate on specific Linux kernel components like file systems and device drivers [2, 24]. The eBPF fuzzer from the `iovisor` project [10] is based on libFuzzer, which compiles the verifier as a userspace application by replacing the kernel library calls with userspace calls. The inputs to the verifier do not follow any grammar of the eBPF instruction set and are randomly generated bytecode. Thus most of the input is rejected by the verifier. The fuzzer is based on the v4.3.0-rc3 kernel, where the code size of the verifier is 2K lines.

Another open-source fuzzer [18], generates random instructions by following the grammar of the eBPF instructions. Each generated program runs in separate QEMU-based virtual machines and does not enforce the semantic rules of the eBPF programs. Thus throughput of the fuzzers is not high since the virtual machines require higher computing power. The main limitation would be ignoring the semantic rules, which means the verifier will not pass generated programs.

There is another category of fuzzers where the fuzzing target expects the inputs to follow a specific grammar like compilers. P4Fuzz [1] is one example of a generation-based fuzzer that creates P4 programs to find bugs in P4 compilers. The test-case generators also successfully find bugs in the compilers through differential testing. Csmith [25] is an open-source test-case generator tool that can generate random C programs which conform C99 standard. It can find many previously unknown bugs in both commercial and open-source compilers.

7 CONCLUSION

The Linux eBPF subsystem allows arbitrary userspace programs to be executed inside the Linux kernel. Given eBPF is widely adopted for its flexibility and easy-to-use programming model, it is necessary to harden the verifier to safeguard the Linux kernel. The proposed LKL-based eBPF fuzzer can be used to find the ALU ranging tracking errors with its structured random instruction generator. The LKL-based fuzzer instances allow high fuzzer throughput and can test many eBPF instructions with limited hardware and time. We evaluated the eBPF fuzzer using Linux kernel v5.10.0. The evaluation results show that our eBPF fuzzer can cover 40% of the verifier code in less than 30 minutes, with two existing ALU bugs identified.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments. This work is partly supported by the US Office of Naval Research (ONR) under grants N00014-19-1-2493 and N00014-22-1-2672 and US National Science Foundation (NSF) under grant CNS 2127491. Any opinions, findings, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of these agencies.

REFERENCES

[1] Andrei-Alexandru Agape and Madalin Claudiu Danceanu. 2018. P4fuzz: A compiler fuzzer for securing p4 programmable dataplanes. *Aalborg University* (2018).

[2] Jake Corina, Aravind Machiry, Christopher Salls, Yan Shoshitaishvili, Shuang Hao, Christopher Kruegel, and Giovanni Vigna. 2017. Difuze: Interface aware fuzzing for kernel drivers. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2123–2138.

[3] MITRE Corporation. 2016. CVE-2016-4557. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2016-4557.

[4] MITRE Corporation. 2021. CVE-2021-3490. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2021-3490.

[5] MITRE Corporation. 2023. Linux eBPF-related CVEs. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=ebpf.

[6] NATIONAL VULNERABILITY DATABASE. 2020. CVE-2020-27194. https://nvd.nist.gov/vuln/detail/CVE-2020-27194.

[7] NATIONAL VULNERABILITY DATABASE. 2022. CVE-2022-23222. https://nvd.nist.gov/vuln/detail/CVE-2022-23222.

[8] Andrea Fioraldi, Dominik Christian Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++ : Combining Incremental Steps of Fuzzing Research. In *14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020*, Yuval Yarom and Sarah Zennou (Eds.). USENIX Association. https://www.usenix.org/conference/woot20/presentation/fioraldi

[9] Google Inc. 2023. syzkaller - kernel fuzzer. https://github.com/google/syzkaller.

[10] The iovisor project. 2019. BPF Fuzzer. https://github.com/iovisor/bpf-fuzzer, Online, accessed 05/03/2023.

[11] kernel.org. 2023. BPF Type Format (BTF). https://www.kernel.org/doc/html/latest/bpf/btf.html,Online, accessed 05/03/2023.

[12] kernel.org. 2023. eBPF Instruction Set Specification, v1.0. https://docs.kernel.org/bpf/instruction-set.html, Online, accessed 05/03/2023.

[13] kernel.org. 2023. eBPF verifier. https://docs.kernel.org/bpf/verifier.html, Online, accessed 05/03/2023.

[14] Joshua Levin and Theophilus A Benson. 2020. ViperProbe: Rethinking microservice observability with eBPF. In *2020 IEEE 9th International Conference on Cloud Networking (CloudNet)*. IEEE, 1–8.

[15] LKL. 2023. LKL: Linux Kernel Library. https://lkl.github.io/.

[16] LLVM. 2023. AddressSanitizer. https://clang.llvm.org/docs/AddressSanitizer.html.

[17] Manfred Paul. 2021. LPE exploit for CVE-2021-3490. https://github.com/chompie1337/Linux_LPE_eBPF_CVE-2021-3490.

[18] Snorez. 2021. eBPF Fuzzer. https://github.com/snorez/ebpf-fuzzer, Online, accessed 05/03/2023.

[19] Alexei Starovoitov, Joe Stringer, and Michael Kerrisk. 2023. eBPF Syscall. https://docs.kernel.org/userspace-api/ebpf/syscall.html, Online, accessed 05/03/2023.

[20] Evgeniy Stepanov and Konstantin Serebryany. 2015. MemorySanitizer: fast detector of uninitialized memory use in C++. , 46–55 pages.

[21] Hao Sun, Yuheng Shen, Cong Wang, Jianzhong Liu, Yu Jiang, Ting Chen, and Aiguo Cui. 2021. HEALER: Relation Learning Guided Kernel Fuzzing. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles (SOSP '21)*. 344–358.

[22] Inc. Tigera. 2023. eBPF Explained: Use Cases, Concepts, and Architecture. https://www.tigera.io/learn/guides/ebpf/, Online, accessed 05/03/2023.

[23] Marcos AM Vieira, Matheus S Castanho, Racyus DG Pacífico, Elerson RS Santos, Eduardo PM Câmara Júnior, and Luiz FM Vieira. 2020. Fast packet processing with ebpf and xdp: Concepts, code, challenges, and applications. *ACM Computing Surveys (CSUR)* 53, 1 (2020), 1–36.

[24] Wen Xu, Hyungon Moon, Sanidhya Kashyap, Po-Ning Tseng, and Taesoo Kim. 2019. Fuzzing file systems via two-dimensional input space exploration. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 818–834.

[25] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*. 283–294.

[26] Michal Zalewski. 2023. american fuzzy lop. https://lcamtuf.coredump.cx/afl/.