Evaluating Symbolic Execution Programs for eBPF

Jingyang Men
Carnegie Mellon University
jingyanm@andrew.cmu.edu

Simon Spivey
Carnegie Mellon University
sspivey@cmu.edu

ABSTRACT

The eBPF ecosystem is highly dynamic and constantly evolving to add new features. Much of this evolution has been pioneered by tech giants such as Meta, Google, and NVIDIA. Although such evolution tremendously benefits the ecosystem, these changes have introduced stability issues for systems using custom eBPF code. This has led to a demand for testing frameworks that can identify bugs in different types of eBPF programs. In order to understand long-term feasibility of these tools, we evaluate existing symbolic execution based test-case generation frameworks for eBPF programs.

Keywords

eBPF, symbolic execution, test case generation

1. INTRODUCTION

Bugs relating to functional correctness can be incredibly challenging to identify without running the program or proper procedures with unit tests. There is currently a need for testing in order to identify with confidence the usability of an eBPF (extended Berkeley Packet Filter) program. As the only existing testing methods within eBPF are manual, it is deemed inefficient and limited. This has spurred significant effort into researching symbolic execution to automate the generation of test inputs[2, 5].

The idea of debugging via symbolic execution involves utilizing symbolic values to represent unknown runtime inputs of programs. By gathering data on potential path constraints of programs that are affected by these symbolic values, it is possible to derive the absolute logical constraints that reflect a set of expected inputs responsible for triggering such behaviors in the program.

One of the tools extensively used in this field is KLEE, a symbolic execution engine that leverages LLVM bytecode to perform rigorous testing. By automatically generating test cases that expose bugs in code, it aims for high coverage with minimal manual effort. Its effectiveness is noted to achieve on average more than 90% of code coverage, which is substantially higher than what manual testing efforts typically yield [4].

These tools may help in analyzing and predicting the performance and security behaviors of eBPF programs before running them in the kernel. This is particularly useful for identifying potential performance bottlenecks or security vulnerabilities in network functions (NFs) that eBPF often handles. By abstracting program inputs as symbolic values, symbolic execution allows for the examination of various program paths and states, contributing to more robust and secure eBPF applications in the community.

Given such potential performance gains and utility provided in symbolic execution, we will investigate performance and coverage of symbolic execution based test case generation frameworks on eBPF programs. We will analyze efforts made to support the correctness of eBPF programs in the community and list out current challenges in building a symbolic execution tool for eBPF programs. By focusing on analyzing existing tools and understanding the principles implementation, we hope to make observations that may lead to potential areas of research for future development.

2. BACKGROUND

2.1 eBPF Overview

eBPF is a Linux kernel technology that enables programs to run securely in the kernel space without changing the kernel source. These programs are triggered by events that pass a specific setpoint in the kernel. While this is similar to LKM (Linux Kernel Module), which allows programs to run in the kernel space, LKMs introduce a lot of risk to a system as untrusted code can freely run in the kernel. eBPF programs address security issues by running any byte-code inside a eBPF VM, which is a sandboxed subsystem of the Linux kernel. eBPF includes a verifier, which enforces safety checks and verifications on code that will be passed down. Byte-code is executed via a JIT (Just-In-Time) compiler.

All eBPF programs are triggered by events, which are captured at hooks when a specific set of instructions are

executed. When these hooks are triggered, the eBPF program will execute, letting one capture or manipulate data. Hooks can be placed in many areas within the kernel, such as system calls, function entry & exit, network events, and kprobes. The diversity of locations, as seen in Figure 1, is a property that makes eBPF so valuable.

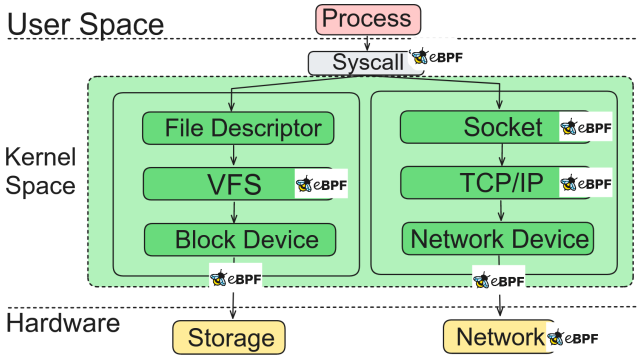

Figure 1: eBPF hook locations

When programs are triggered, eBPF makes calls to helper functions. These functions are defined by the kernel, which makes eBPF feature-rich in accessing memory. Common examples of helper functions include: key-value pairs in tables, manipulation of network sockets, metadata, and chaining eBPF programs together (which is commonly called tail calls).

There are many applications developed utilizing eBPF concept, the majority of them are XDP (eXpress Data Path) based programs - meaning they run as hook points in the NIC. Some commonly used programs include Katran and Crab. **Katran** [17, 12] is a C++ library and BPF program to build high-performance load balancing forwarding planes. Katran leverages XDP infrastructure from the kernel to provide an in-kernel facility for fast processing of packets. **Crab** [15] provides another alternative for load balancing schemes. Crab aims at eliminating latency overheads and scalability bottlenecks while simultaneously enabling the deployment of complex, stateful load balancing.

2.2 eBPF Architecture

Following Figure 2, starting at the bottom left at '1' labeled: "eBPF Program":

1. eBPF programs are mostly written in C. There are very specific limitations defined in the `bpf.h` and `bpf_helpers.h` header files that define specific BPF functions, structs, and functions to interface with BPF maps - eBPFs preferred key-value data structure. The most common library used to help compile is `libbpf`, which provides easy to use library APIs for applications.

2. Under the hood, eBPF uses LLVM as its compiler to compile C code into bytecode. The kernel expects all programs to be loaded as bytecode.

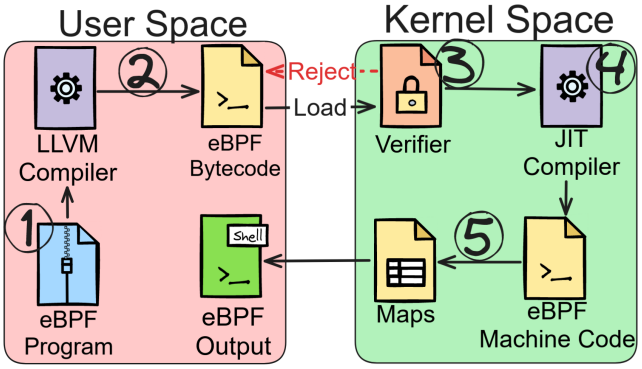

Figure 2: Architecture

3. The in-kernel verifier will make sure that code passed in passes a series of checks. It simulates the program in a sandboxed VM to confirm that the program passes a series of checks for safety. The program must not infinitely loop, access unprivileged data, or crash the kernel.

4. If the program passes all the safety checks, the JIT compiler compiles the code into native machine code. This fast compilation allows eBPF to remain relatively quick, even though it does verification checks inside a VM.

5. The `bpf()` system call will then load it onto the specified hook point. The program waits for an event to trigger itself. There are a plethora of hook points available where data is collected from the eBPF programs to be loaded into maps. Maps are the bridge between sharing data between eBPF programs, or eBPF programs and user space.

2.3 Symbolic Execution and Applications

Symbolic execution is a common technique used to find bugs within software. It is a form of formal verification that explores multiple control flow paths within a program.

Instead of executing code with inputs that are either manually or randomly prepared, these tools execute code with symbolic inputs. They replace actual program inputs with symbolic values and perform operations that handle these symbolic values instead of concrete values.

Referring to Figure 3, this simple function has a large range of acceptable inputs that may be tedious to create unit tests to assure complete validation. With symbolic execution, we reduce down the conditional statements into path conditions. Substituting x^2 for area, The final path conditions of this code simplify to $x^2 = y$ and

```
void square(int x, int y) {
    int area = x*x;
    if (area == y)
        if (x > 2y)
            ERROR;
}
```

Figure 3: Symbolic execution pseudo code

$x <= 2y$. The symbolic execution framework then attempts to find the ranges that satisfy both path conditions. If a satisfying range of x and y are found, it attempts to pick concrete numbers within that range to confirm that the *ERROR* state is reached. While this technique is excellent at identifying bugs, it is difficult to scale to larger programs and needs to be intentionally optimized in order to be practical.

Two foundational symbolic execution frameworks are EXE and KLEE[6, 4]. If a statement uses a symbolic value, EXE adds it as an input-constraint while keeping other statements run as usual.

KLEE, on the other hand, built an interpreter loop which "selects a state to run and then symbolically executes a single instruction in the context of that state" [4]. This loop continues until there are no states remaining, or a user-defined timeout is reached.

Other publications utilizing the concept of symbolic execution are PIX and GenSym, which all build on top of KLEE [13, 18]. PIX aimed at building an interface that can measure performance of network functions. By digging into their repo, all their network functions refer to eBPF programs. GenSym aims at building a faster LLVM compiler, which takes built source code and output generated C/C++ code that allows symbolic execution to run faster and generate broader coverage.

3. EXPERIMENT DESCRIPTION

3.1 Problem

It is very difficult to write completely validated code in the kernel. The only thing preventing malicious or faulty code running the kernel is the verifier. While the verifier is a great tool, it enforces kernel-level safety guarantees at the userspace level - meaning it is not able to catch many issues. There are existing issues in both core eBPF functionality and popular eBPF programs.

For example, there have been numerous bug reports of kprobes not invoking when they should. These issues are very difficult to diagnose when reproducibility is sporadic, at best. Probes are also prone to event overloads, which happen when event producers spamming the eBPF hook. eBPF has no concurrency primitives to block events. This leads to data loss and corruption. [16] There are mitigation strategies for these issues, but these issues are difficult to reproduce. In order to discover these issues, symbolic execution attempts to discover these lurking bugs.

eBPF programs are usually compiled more than once while a system is running - which can make re-testing programs a very expensive job. There are multiple frameworks used in order to achieve symbolic execution, but no comparative analysis between frameworks. How should a symbolic framework for eBPF be evaluated? We aim to evaluate performance of existing tools while understanding how each framework uniquely searches for bugs in order to discover potential research paths forward.

3.2 Test targets

There are 2 targeting frameworks that we have selected: eBPF-SE/PIX and eBPF-Equivalence-Check. eBPF-SE is the newer version of PIX, which supports newer version/release of Linux. The PIX framework only supports old Linux releases (such as Ubuntu 16.04 and earlier). eBPF-Equivalence-Check is a framework that aims to compare two eBPF programs. This framework is designed to compare if functionality has changed between versions of an eBPF program.

For both of these frameworks, we will focus on the coverage and time consumption of these test cases. We will use SE to represent eBPF-SE and EQC to represent eBPF-Equivalence-Check.

3.3 Benchmarking

Benchmark programs will be using XDP programs. XDP refers to a high-performance programmable network data path in the Linux kernel. XDP programs are part of a larger technology stack that allows for the efficient processing of packet data directly at the lowest level of the software network stack. Most XDP-programs aim at implementing complex functionalities, which are used to locate higher layers of network stack closer to network hardware.

Socket-op eBPF programs are attached to socket operations, providing a mechanism to execute custom logic at various stages of the socket lifecycle or during specific socket-related operations. These programs are particularly useful for socket filtering, network monitoring and performance probing.

XDP programs selected for the Benchmark include:

- crab - load balancer from the CRAB project [15].
- fw - firewall from the hXDP project [3].
- katran - load balancer from Facebook [12].
- fluvia - IPFIX Exporter from the Fluvia project [8].
- hercules - High speed bulk data transfer application from Network Security Group at ETH Zürich [1].

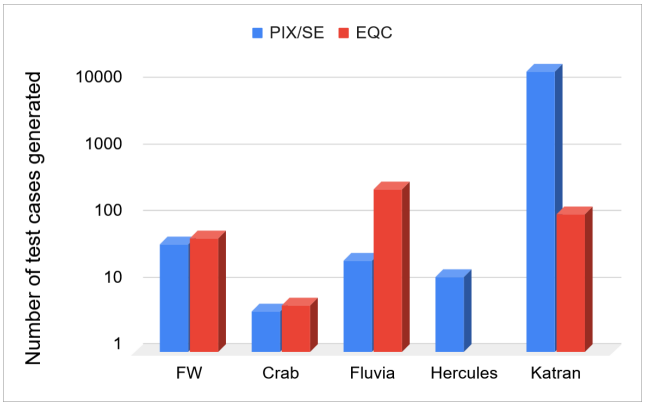

Figure 4: Number of case tests generated for each framework against XDP programs

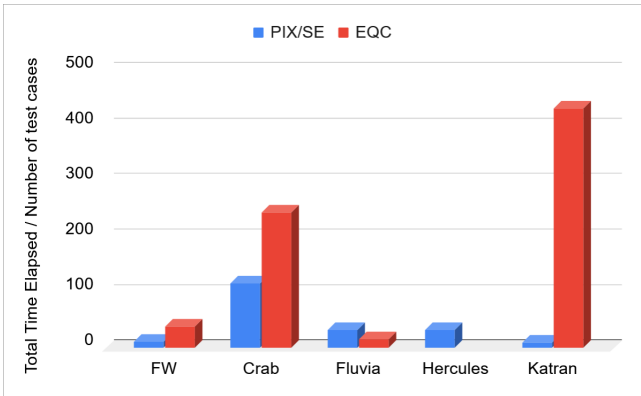

Figure 5: Time Elapsed per test case for each framework against XDP programs

These programs are selected for the benchmark because they are programs offered in both eBPF-SE and eBPF-equivalence-check repositories. There is some guarantee that these frameworks support these programs so that we have confidence in the test results.

3.4 Environments and metrics

We run these 2 frameworks in docker containers on a local machine. Each container is allocated with 4 cores of 2.3 GHz Quad-Core Intel Core i7 processor and 16 GB of memory. Container images are pre-built following instructions from their repository respectively.

For each framework on each benchmark program, we use two metrics: number of test cases generated and time per single test case. This aligns with metrics used in existing symbolic execution test case frameworks [18, 4, 13]. Running each benchmark program will give a number of test cases generated and time elapsed. We simply divide time elapsed by the number of test cases to get time per single test of a benchmark program.

4. EVALUATION

The layout of the evaluation section is: we first test the chosen framework on coverage and efficiency. (section 4.1) and then theorize based on the data from the results and by digging into source code of the two frameworks (section 4.2).

4.1 Results

Both frameworks fail to support non-xdp programs, so we will only show results against XDP variants of the programs in figure 4 and figure 5. Figure 4 shows the number of test cases generated by SE and EQC. From the figure we can deduct EQC is more thoroughly testing FW, Crab and Fluvia, yet generates about 10% less test cases than SE in Katran. EQC also failed to generate test cases for Hercules.

Figure 5 shows time elapsed per test generation and neither framework dominates. Among benchmark programs, Katran-SE presents a very special outlier: generating more than 10 times more test cases while keeping time per test less than a magnitude vs EQC (8ms vs 430 ms).

```
void *map_allocate(char* name,
char* key_type, char* val_type,
    unsigned int key_size,
    unsigned int value_size,
    unsigned int max_entries) {
  //Allocate Map Structure
  struct MapStub *map = ...
  klee_assert(map->keys_present
    && map->values_present);
  klee_make_symbolic(map->values_present,
    max_entries*value_size, map->val_type);

  for (int n = 0; n < NUM_ELEMS; ++n) {
    //initialize map entries
  }
  return map;
}
void array_reset(struct ArrayStub *array){
  // reset array
  array = ...
  klee_make_symbolic(array->data,
    (array->capacity * array->value_size),
    array->data_type);
}
```

Figure 6: Symbolic trigger of SE in pseudo code

4.2 Observations

To our surprise, we found that the number of test cases generated per eBPF program significantly differed, with no noticeable trend. Before suggesting any theories, we need to look at the underlying logic of both

```
1  void *map_lookup_elem(struct MapStub *map,
2      const void *key) {
3
4    /* Generating symbol name
5       and do map lookup*/
6    char *final_sym_name = ...
7    int map_has_this_key =
8      klee_int(final_sym_name);
9
10   if (map_has_this_key) {
11     //generate return value string
12     map->key_deleted[map->keys_seen] = 0;
13     map->keys_seen++;
14     char *ret_value_str = ...
15
16     void *ret_value =
17         malloc(map->value_size);
18     klee_make_symbolic(ret_value,
19         map->value_size, ret_value_str);
20
21     return ret_value;
22   } else {
23     map->key_deleted[map->keys_seen] = 1;
24     map->keys_seen++;
25     return NULL;
26   }
27 }
```

Figure 7: Symbolic trigger of EQC in pseudo code

frameworks. By comparing source code, we found the major difference between SE and EQC lies on different triggers of a symbolic execution path. Both EQC and SE used an array to emulate changes to map during eBPF runtime.

As shown in Figure 6 line 10 - 23, a new symbolic execution path is triggered when array is reset or a new map array being allocated (by calling `malloc()`) in SE, making the test cases generated by SE only sensitive to map creation and reset. However, this approach completely omits any changes during eBPF runtime. In EQC, the function shown in Figure 7 on line 18, a new symbolic execution only can be triggered whenever a map entry is queried, making EQC more sensitive to map array usage during runtime instead of map array creating and reset.

While EQC claims to explore more paths of a program, this is questionable when we investigate source code of Katran. Katran initialized 9 different maps during initialization while other programs such as fw uses less than 4 maps. This difference could explain the surge of the number of test cases in the Katran-SE couple, since the number of different combinations of 9 factors should be much more than 4 factors.

This differing in test case generation has led to inconclusive results regarding which framework is better. We cannot make any conclusions about performance until there is an understandable reason why the number of test cases wildly differ. There is a need to research more to better understand which method provides more value to a user trying to test eBPF programs, before we can fairly compare frameworks against each other.

5. FUTURE RESEARCH

While we did not include non-XDP programs due to lack of support with the frameworks selected, these programs make up a notable chunk of all eBPF programs. Non-XDP programs include traffic control eBPF programs and socket-op programs. Traffic control eBPF programs are used to manage and control the traffic on network interfaces within the Linux kernel. These programs are typically attached to the traffic control (tc) subsystem of the Linux networking stack. The primary use cases for tc eBPF programs include packet scheduling, packet prioritization and queue management.

We hope to research further into Non-XDP programs, as the frameworks are already working on bringing in a select few:

- dae - proxy from the daeuniverse project [9].
- falco - kernel monitoring agent from the Falco project [11].
- rakelimit - multi-dimensional fair-share rate limiter designed for UDP by Cloudfare [7].

In addition, with the Rust programming language receiving some adoption in the linux kernel, there has been a call in the eBPF community to use the Rust ecosystem to replace the LLVM compiler. With Rust being a memory safe language, this will likely help reduce errors of eBPF programs loaded onto the system. While test Rust replacement frameworks currently exist, it would be interesting to observe any change in performance of symbolic execution tests and if the trend of the number of total tests changes between languages [10].

Recently, The linux eBPF community has released a command that takes a simpler approach to better debug eBPF programs, `BPF_PROG_RUN`. This is a command that provides the user with the ability to utilize unit tests against their eBPF programs before needing to load them into the verifier. This will help provide more confidence to developers writing eBPF programs. There is potential to leverage this command with symbolic execution test cases, by generating symbolic inputs for this wrapper instead of a virtualized runtime. This may lead to better optimized testing[14].

6. RELATED WORK

GenSym [18] aims at building a faster LLVM compiler, which takes built source code and output generated C/C++ code that allows symbolic execution to run faster and generate broader coverage. GenSym does not support eBPF programs currently, but we are trying to extend the functionalities of GenSym and make it compatible with eBPF programs.

In our experiments, GenSym can support many normal libc functions such as link, dirname and pathchk. From the benchmark offered in the GenSym repository, we found GenSym also supports programs with multiple C files and header files. Based on this, we argue that it is possible to make changes to GenSym to support eBPF programs.

According to our meeting with the GenSym team, the reason GenSym reports errors to eBPF programs is the lack of BPF libraries in GenSym translator and use of unsupported compile command lines. A solution is to compile the customized libbpf into .ll object code and avoid using advanced functions such as debugging messages during compilation.

7. CONCLUSIONS

In this study, we tested and compared coverage of existing symbolic execution based test case generation frameworks. We found that our results are somewhat inconclusive due to a surprising, non-deterministic difference in test cases generated per each framework. We analyzed the source code of these frameworks and it has sparked questions on the differing methods of generating test cases. We are arranging meetings with the authors to understand the difference in methodology and how to properly measure the effectiveness of the tools.

8. REFERENCES

[1] N. S. G. at ETH Zürich. Hercules, high speed bulk data transfer application., 2024. Last accessed 18 April 2024.

[2] P. Boonstoppel, C. Cadar, and D. Engler. Rwset: Attacking path explosion in constraint-based test generation. pages 351–366, 03 2008.

[3] M. S. Brunella, G. Belocchi, M. Bonola, S. Pontarelli, G. Siracusano, G. Bianchi, A. Cammarano, A. Palumbo, L. Petrucci, and R. Bifulco. hxdp: Efficient software packet processing on fpga nics. *Commun. ACM*, 65(8):92–100, jul 2022.

[4] C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of High-Coverage tests for complex systems programs. In *8th USENIX Symposium on Operating Systems Design and Implementation (OSDI 08)*, San Diego, CA, Dec. 2008. USENIX Association.

[5] C. Cadar and D. Engler. Execution generated test cases: How to make systems code crash itself. pages 902–902, 08 2005.

[6] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: Automatically generating inputs of death. *ACM Trans. Inf. Syst. Secur.*, 12(2), dec 2008.

[7] cloudfare. Rakelimit: A fair-share ratelimiter implemented in bpf., 2024. Last accessed 18 April 2024.

[8] N. Communications. Ipfix exporter using ebpf/xdp and ipfix library in go, 2023. Last accessed 18 April 2024.

[9] daeuniverse. Dae:a high-performance transparent proxy solution., 2024. Last accessed 18 April 2024.

[10] A. Decina. Aya: an ebpf library for the rust programming language, built with a focus on developer experience and operability., 2024. Last accessed 28 April 2024.

[11] falco. Falco: A cloud native runtime security tool for linux operating systems, 2024. Last accessed 18 April 2024.

[12] M. Incubator. Katran: A high performance layer 4 load balancer, 2024. Last accessed 18 April 2024.

[13] R. Iyer, K. Argyraki, and G. Candea. Performance interfaces for network functions. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 567–584, Renton, WA, Apr. 2022. USENIX Association.

[14] T. kernel development community. Running bpf programs from userspace., 2024. Last accessed 28 April 2024.

[15] M. Kogias, R. Iyer, and E. Bugnion. Bypassing the load balancer without regrets. In *Proceedings of the 11th ACM Symposium on Cloud Computing*, SoCC '20, page 193–207, New York, NY, USA, 2020. Association for Computing Machinery.

[16] M. Leogrande. kretprobes are mysteriously missed, 2020. Last accessed 28 April 2024.

[17] R. D. Nikita Shirokov. Open-sourcing katran, a scalable network load balancer.

[18] G. Wei, S. Jia, R. Gao, H. Deng, S. Tan, O. Bračevac, and T. Rompf. Compiling parallel symbolic execution with continuations. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1316–1328, 2023.