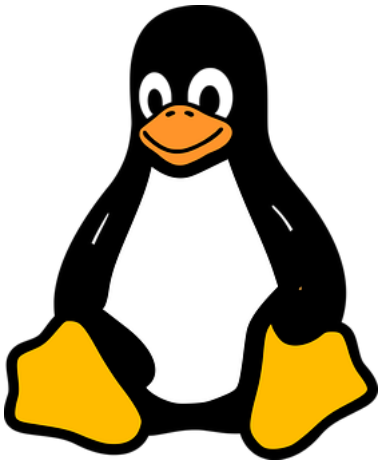
Sorbonne Université - Polytech Sorbonne

Technical Internship Report

MAIN 5

Worst-case Time Behavior of BPF Programs



Student:
Mahshid Khezri Nejad

Tutor:
Paul Chaignon

29/08/2019

Résumé en français

Ce document présente mon travail de 5 mois à Orange Labs, Châtillon. Le projet que j'ai mené sous l'encadrement de Paul CHAIGNON a pour objectif la conception d'un nouvel outil d'analyse de programmes. En particulier, cet outil doit permettre de générer des entrées potentielles d'attaques sur une nouvelle technologie présentée du noyau Linux (BPF), utilisée pour la protection et maintenance de la sécurité du réseau.

Notre problématique s'inscrit dans le domain de l'Analyse de la fiabilité des logiciels (*Software reliablity analysis*). Nous avons commencé par dresser un état de l'art des méthodes et approches potentielles applicables à notre problématique. Notamment, nous avons étudié l'exécution symbolic, fuzzing et les méthodes hybride.
Ensuite, nous avons décidé de baser notre approche sur l'execution symbolic, qui exécute le programme symboliquement, i.e., il assume des valeurs symboliques pour les entrées du programme et exécute le programme avec ces entrées symboliques.

Afin d'appliquer cette méthodologie à notre problématique, notre outil génère un graphe de flow des instructions du programme et les exécute symboliquement en cherchant le chemin qui prend le plus de temps à s'exécuter. Pour trouver ce chemin, nous avons choisi l'heuristique de A*, qui demande une estimation de "coûts" de chaque instruction BPF.

Nous avons commencé à développer un moteur d'exécution symbolique pour atteindre notre but. Pour l'instant, notre outil détecte les instructions BPFs et nous avons une estimation de temps d'exécution pour quelques instructions particulières à BPF. Pour la suite de travail, il nous reste à estimer le temps d'exécution pour les autres instructions BPF et les ajouter à l'outil pour ensuite être utilisées par l'heuristique.

I would like to thank my professors at Polytech-Sorbonne who helped me learn better ways to solve new scientific problems throughout my three years as an engineering student, specifically Hacène Ouzia, with whom I did 2 optimization projects, as well as Frédérique Charles and Thibault Hilaire who interviewed me three years ago for admission and accepted me as a MAIN student.
I would also like to thank my internship tutor, Paul Chaignon who introduced to me a field of Computer Science which I did not know before, and guided me through this internship to learn the necessary knowledge and acquire several new skills. Also, Sébastien Allard, who as a manager, is always eager to build a better workspace for everyone including interns like myself. I would like to thank Xiao Han who was always helpful with technical problems or challenges alongside my way. I would also like to thank Kahina Lazri and Yoann Ghigoff for the lively discussions and games during break times.

Contents

1 Introduction

In this document I will present my final internship, which took place from March 25th to September 24th 2019, at Orange Labs in Châtillon, under the supervision of Paul Chaignon, Research Engineer. Orange Labs is the research division of Orange and is dedicated to carrying out the company's technical innovational research tasks.
My internship is a research internship which consisted of the following stages:

- State-of-the-art review
- Reflection to choose the appropriate approach
- Development

I worked autonomously under the lead of my supervisor.

Before getting into context and details of our work, I would like to give a short explanation on the interest of this work for the company and more particularly the security department.
Currently, network functions isolation is realized by hardware-based mechanisms for memory isolation. However, there exists software-based alternatives for network isolation. Among these alternatives, there is BPF which is a way to extend Linux kernel and can be used to isolate network functions. Despite the fact that BPF has been added to Linux kernel since 2014, there are still some unstudied aspects regarding BPF. Our work contributes to understanding the behavior of a given BPF program. If an BPF program takes too long to execute, it can cause a denial of service in several components of Linux kernel. Knowing this worst-case execution time and the inputs that trigger it helps Orange better predict and avoid problematic situations.

1.1 General Context

Software Reliability Testing is a field that helps discover many types of problems in Software Design or a software's functionalities. The aim of this work is also to propose a tool to test reliability of a specific program type, BPF. More precisely, we want to know the longest time that our program might take to execute. We also want to know what inputs make our program take this longest execution time.

BPF is a new technology introduced in Linux kernel version 3.15. It can be used to add code to the kernel from user space in order to do various tasks such as monitoring a specific function or filtering packets.

We will continue by explaining our problematic in more details, then give a brief introduction to BPF and afterwards talk about the different existing approaches to solve these kinds of problems as well as the related works.

1.2 Problem Presentation

Given a specific type of program, BPF, we would like to find all the inputs to this program that trigger its worst-case execution time. To achieve this goal, we need to consider algorithmic complexity of the given program, as well as more practical aspects such as, microarchitecture and operating system effects.
Let's start by reminding the notions of *asymptotic algorithmic complexity* and then proceed by being more specific about other factors influencing the execution time programs.

Asymptotic Algorithmic Complexity:
Asymptotic algorithmic time complexity describes the relationship between the size of a given input to a program and its execution time, only by looking at algorithmic aspects and ignoring characteristics of the machine. The worst-case complexity is the worst-case scenario considering asymptotic algorithmic complexity of a program. Remember that an algorithm's behavior can actually vary depending on the input.

Consider quick-sort as an example: Quick-sort's average time complexity is $O(nlog(n))$; however, if we pass a reversely ordered array of integer numbers to quick-sort, it's execution time will be of $O(n^2)$.

Other Aspects:
As said before, the worst-case execution time of a program doesn't only depend on its asymptotic algorithmic complexity. To estimate the real worst-case execution time, we need to consider microarchitectural aspects such as:

- CPU cache misses and the speed difference among different levels of cache.
- Branches and pipelining.
- Scheduling and I/O.

We cannot treat all memory operations equally due to cache misses and speed difference in different levels of cache. Meaning that, for example, when calculating the worst-case execution time, we cannot consider a constant c for all memory operations; since some may take longer or faster to return.
Besides, if branch prediction is poorly executed, it can slow down execution of program instructions. Furthermore, with pipelining the execution time of an instruction depends on the other instructions around it. Branches and pipelining might also impact the order of fetching instruction and as a result some instructions may take longer to be fetched.
In addition, let's not forget that our program might interact with devices, and be on hold until an I/O operation is finished.

Returning to our problematic, we can in short say that our problem consists of finding the "longest" program path, considering its algorithmic complexity *and* its interactions with the machine and the operating system.

2 Background

2.1 Brief Review on BPF

BPF is an extension of classic BPF (BSD Packet Filter) introduced in 1992 as a new kernel architecture to accelerate the packet filtering process in Unix.
BPF can be seen as a virtual machine to extend the Linux kernel in a safe way from user space, eliminating the need of writing separate modules. BPF has its own bytecode which is verified by the Linux kernel before being loaded into the kernel to ensure that the BPF program running in the kernel cannot harm the system, contrary to hand-written modules. BPF programs can be attached to kernel events such as packet reception or different functions for performance analysis. Precisely, BPF programs are attached to a designated code path in kernel, meaning that each time that path is executed, the attached BPF program will be executed.

Program Types

Each BPF program has a specific program type, related to its use. For example socket related program types are:

- BPF_PROG_TYPE_SOCKET_FILTER: Can be used for filtering actions, such as, dropping or trimming packets.
- BPF_PROG_TYPE_SOCK_OPS: Can be used to catch socket operations like connection establishment.
- BPF_PROG_TYPE_SK_SKB: Can be used to to access socket details such as port and IP address.

Data Structures

BPF has its own specific data structure: BPF maps. BPF maps are data structures allocated in the kernel's memory space that can be shared between different BPF programs as well as user-space programs: Figure 2. Twenty-four different map types have been defined until now to support a variety of functionalities such as, hash-tables, arrays and stacks.

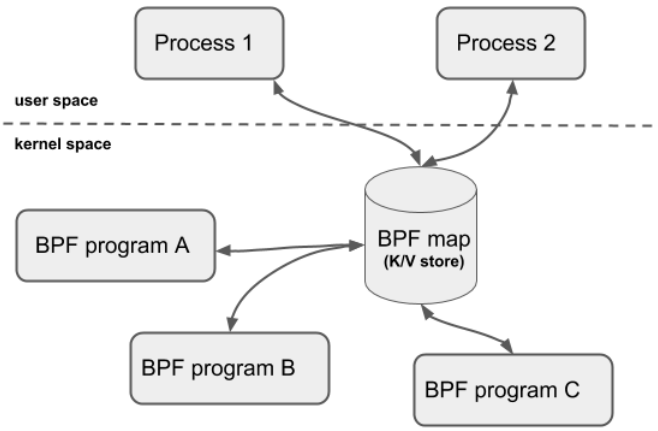


Figure 2: Access to BPF maps (from **[Cilium]**)

BPF Helpers

BPF helpers are simply *functions* to extend the capabilities of the BPF virtual machine. Each different program type has a specific set of helpers and all BPF program types share a basic set of helpers that ensure basic operations such as looking up an element in a map or deleting one. The common set of helpers can be found in the appendix 6.

Compilation and Execution

As mentioned in the introduction, BPF has its own bytecode and instruction set but the LLVM Clang compiler has added support for BPF and can compile C into BPF bytecode. Object files that contain this bytecode can be directly loaded into the kernel by a specific system call. This bytecode then needs to be compiled into the host machine's assembly code. To this end, there is a just-in-time (JIT)[1] compiler for BPF bytecode in kernel.

Verification

Before being loaded, BPF code is verified by a static analyzer in Linux kernel. This verifier ensures that the program does no unsafe actions, such as memory access; it also verifies that the BPF program terminates by not permitting programs with more than 4096 instructions and out-of-bound loops[2].
BPF verifier first builds a Directed Acyclic Graph from the control flow of the program, then performs a Depth-First Search on this graph to ensure there are no cycles, and no unreachable or unsupported instructions. Then, to ensure the safety of memory accesses and correctness of arguments passed to helper functions, the verifier re-traverses all possible paths in the graph.
The verifier needs to be fast since it is launched each time a BPF program is loaded into kernel.

Program Chaining

To overcome the instruction number limitation imposed by verifier we can *chain* BPF programs BPF tail calls. BPF tail calls are a way to call a BPF program from another program without returning to the first. Its implementation is different from normal function calls since it is implemented like a long jump on the same stack frame. Although BPF tail calls could be considered a way to extend the BPF code to execute, there is a limit of 33 on the number of total calls permitted by kernel.

[1] JIT compiling is a way of executing computer code that involves compilation during execution of a program (at run time) rather than prior to execution. ([**16**]

[2] Since we started this project, a minimal support for bounded loops have been added to the Linux kernel.

2.2 Software Analyzing Approaches

The aim of this project is to find the worst-case execution time of BPF programs and the *inputs* which trigger this worst-case execution time.
We have studied three different software analyzing approaches which have been used in many tools and articles to find software bugs, or to study software behavior. These methods are as follows: symbolic execution, fuzzing and hybrid methods.

We specifically chose these three methods because they provide the inputs that trigger a certain behavior, which is exactly what we are looking for.
In this section, starting from symbolic execution, then fuzzing and then hybrid methods, we will briefly explain each of these methods, and discuss their advantages and disadvantages.

2.2.1 Symbolic Execution

2.2.1.1 Introduction

Let's start by giving a short introduction to symbolic execution. symbolic execution is a means of analyzing a program to determine the inputs that cause specific parts of a program to execute. It is mostly used to detect bugs, as well as, the inputs that trigger those bugs in a software.

In symbolic execution, symbolic values are assumed for input variables and program instructions are executed based on these symbolic values. Each time a conditional instruction is executed in symbolic execution, a constraint is added on the symbolic values to either continue with the true branch or the false branch. So, for each instruction, based on the constraints to reach this instruction, we will have a set of symbolic constraints that trigger a path in the program leading to it. Finally, by passing this set of obtained constraints to a mathematical solver like *z3*, we can obtain real-world inputs that triggers the parts we want in the program.
A symbolic execution process can be briefly described as follows:

1. Symbolic values are assumed for the program's inputs and unknown variables.
2. An interpreter follows the program flow with the assumed symbolic values.
3. Each distinctive path in the program gives us certain constraints on the assumed symbolic values.
4. By passing the constraints of interesting paths to a solver, we can then get real world inputs to pass to the program.

Let's consider the following example to understand this process better. Assume we have the following C code:

```
0 int foo(int a, int b)
1 {
2     int b = 24;
3     if(b % a == 0)
4     {
5         a -= 2;
6     }
7     return b / a;
8 }
```

We can clearly see that if a equals 2, we will pass by the condition at line 3 and we will get a *Division by zero* error at line 7. Also, if a equals 0, we will get another error at line 3 when checking $b\%a$.
The corresponding control flow graph would look like the following graph:

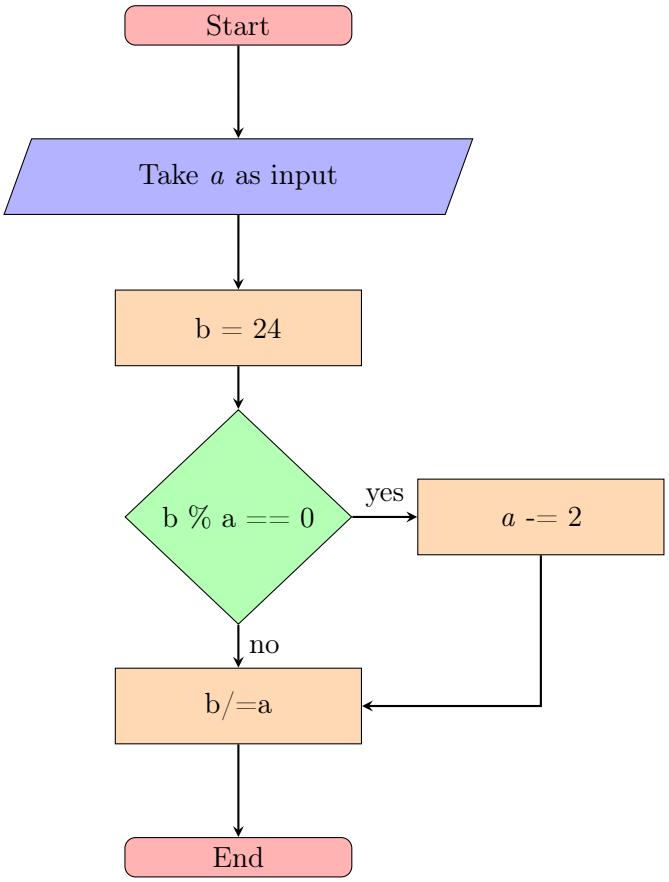


Figure 3: Flow chart

At first, the symbolic execution analyzer considers a symbolic value for a, let's say α and then adds the constraint $\alpha|24$ for the lines of code following the true case of *if* condition at line 3. To get a *division by 0* at line 7, we either have passed by the instructions followed by the true case or not; for the former, $\alpha - 2 = 0$ constraint is also added to set of constraints and for the latter case we have $\alpha = 0$. At the end, we will get:

$$\begin{cases} 24\%\alpha = 0 \\ \alpha - 2 = 0 \end{cases} \text{ or } \alpha = 0.$$

By passing the two constraints on the left to a solver, we get $\alpha = 2$. Now we have two input values that cause an error.

2.2.1.2 Advantages and Disadvantages

The of number possible paths in a program can grow exponentially. And since we try to explore *all the possible paths* in a program, the main problem we can face when running a symbolic execution tool is path explosion. Path explosion can considerably slow down the symbolic execution process to the point that the method loses its practicality.
The other problem is more a technical one. In a real-world program we normally use several

libraries and different frame-works. How are we going to treat them in symbolic execution? Another issue is use of pointer variables. Do we need to symbolize the pointer itself in addition to the memory space pointed to? In what cases?
Considering our use case, if we want to use symbolic execution to find the worst execution path, that is, the longest execution path, we will also need to have an estimation on the cost of each instruction. As a result, characteristics of the machine (like cache) we are running the program on, become important.

Although, as mentioned before, the practicality of use of a symbolic execution tool on a real-world software that can consist of millions of lines of code with many interactions with the environment may be under question, we can be sure that symbolic execution will eventually cover all program paths.

2.2.2 Fuzzing

Fuzzing is another technique to analyze a program's behavior. It is mostly used to expose program vulnerabilities and security bugs. It consists of passing massive amounts of random inputs to a program, then monitoring the program's behavior on these inputs to look for certain things like bugs or memory crashes.
The fuzzing process can be described by the following steps:

1. Generate a random seed of input data.
2. Pass this data set to the program.
3. Find out what parts of the program or what specific behaviors such as, different bugs or were triggered by each input samples.
4. Keep the "*good*" input samples and mutate them to get a new set of data.
5. Go back to 2.

A fitness function or a predefined heuristic determines the definition of *good* in step 4. This fitness function can vary depending on how much information the fuzzer has from the program, or depending on the objective we are trying to reach: whether it is to search more parts of the program to find bugs or to find the inputs that take a considerable amount of time to process. *Mutation* step can consist of different mutation operations, such as adding or removing a byte from input, randomly modifying a bit in the input, randomly changing the order of a subset of input bytes and *etc.*

Fuzzing tools may be categorized into 3 groups **[6]**: White/grey/black-box, depending on whether it is aware of program structure, which can affect its fitness function.

2.2.2.1 Advantages and Disadvantages

Fuzzers are rapid tools that give immediate results, but they are not as reliable as Symbolic-Execution-based tools since their performance depends greatly on the quality of the first input seed. Since the mutation operations are random, if the first seed is poorly chosen, then it might take us a considerable number of mutations to find inputs that could have been obtained by far fewer mutations.

2.2.3 Hybrid Methods

Different hybrid methods have been proposed to take advantage of fuzzing and symbolic execution's positive points and palliate their disadvantages. To go further, we need to know what *concolic execution* is. Concolic execution consists of concrete execution alongside a symbolic one on a sample input.
Hybrid methods can differ in various ways:

- The functionality of their fuzzers
- Do they use concolic execution? If so, how?
- How they mix fuzzing with symbolic execution: e.g. is it iterative? symbolic execution first, then fuzzer?

2.2.3.1 Advantages and Disadvantages

Although Hybrid methods try to take advantages of both methods, they can never be as certain and comprehensive as symbolic execution tools. Also, we may design a complicated hybrid method which, in the end, won't be more efficient as fuzzing alone or symbolic execution run solely.

3 Related Works

In this part we will review other works and articles which attempt to solve the same problem as ours, which is to find a set of inputs that would make a program take its worst execution time path.
We will first discuss symbolic execution related methods and articles, then proceed with fuzzing approaches, and finally, discuss hybrid methods.

3.1 Symbolic Execution

The main problem we can face when running a symbolic execution engine is path explosion which considerably slows down the symbolic execution process. Also, to use symbolic execution to find the worst-case execution path, we also need to have an estimation on the cost of each instruction.So here, the characteristics of the machine we are running the program on become important. Another thing to consider is hash functions and how to treat instructions with hash calculations or hash table lookups in symbolic execution. The following works have both proposed different approaches to overcome the path explosion problem and they use symbolic execution to find the inputs that trigger the worst-case execution time.

- **Castan [1]** takes the LLVM code of a program, for example a network function, and a processor specific cache model as its input. It then tries to discover execution paths that consume large numbers of CPU cycles and synthesizes workloads that trigger those paths. There are 3 interesting points in this work:

  1. Modeling L3 cache: They reversed engineer and modeled the L3 cache of their machine (CPU: Intel Xeon E5-2667v2) in order to have a prediction of cache misses to better simulate the cost of memory access operations.
  2. Handling hash functions: Each time there is a call to a hash function, the result is stored as a symbol different from the initial input symbols. The symbolic execution then continues its routine, and in the end, by using a constraint solver they can find a possible candidates for the symbolic values corresponding to hashed values. Next, to replace these symbols with an expression of the initial input symbols, they inverse the hash function by using rainbow tables [**15**]. Finally, after finding hashed symbolic values, they can continue with the rest of the symbolic execution process.
  3. Notion of *potential cost* and deciding which branches to continue: They have also introduced a notion of *potential cost* for each node, which is the estimation of what it would cost (execution time estimation) if we follow the branch starting from this node. They have attributed a *total cost* to each node which is the sum of this potential cost and a *current cost* which is an estimation of the cost to arrive to this node from the root. When running symbolic execution, in order to avoid path explosion, they use A* to find the worst case path using the aforementioned notion of total cost.

  Castan seems like an interesting work, however, there isn't any information on its execution time in the evaluations.

- **WISE [2]**: The main idea behind this article is that if an input of size N triggers the worst case behavior for all inputs of size N, by taking the path p, then an input of size $M > N$ that triggers a path q with p as prefix would be close to the worst case of all inputs of size M.
  According to its evaluations, WISE's performance is limited to simple programs using one algorithm or one data-structure such as a sorting algorithm or an AVL tree, and it would

take much longer time (more than 3 hours) on more complex applications to give results, that it might seem less practical on real world complex programs.

3.2 Fuzzing

The works we have looked into in more details were mutation-based. It means that after passing a generation of inputs the program was monitored and the inputs which seemed to have maximized a predefined heuristic were then kept and mutated for newer generations.
The first studied tool was SlowFuzz [**3**] and then we continued with PerfFuzz [**4**] which seemed to have a more developed heuristic function.

- **SlowFuzz** [**3**]: SlowFuzz randomly selects an input to execute from a given seed corpus which is mutated and passed as input to the tested program. During program execution, profiling information is gathered, and a calculated *score* is given to the passed input based on its resource usage. The inputs which seem to slow down the program will then be added to the mutation corpus. The afore-mentioned "score" is calculated by a fitness function that keeps track of the total count of all instructions executed during one run of the program on an input. The authors believe that the total instruction count is a good measure to find adversarial workloads for a program.
  This measurement could be useful when considering only the asymptotic algorithmic complexity since we know that we cannot treat all instructions the same way, for example, cost of a memory access instruction is greater than the cost of an addition instruction.

- **PerfFuzz** [**4**]: It follows the same routine as SlowFuzz with one difference: instead of looking at total execution count of instructions, they keep count of how many times a control flow graph edge has been executed and favor the inputs that maximize the execution count of at least one control flow graph edge. To be more precise, if an input results in more code coverage or maximizes an edge execution count for some component then it is added to the set of inputs to be mutated. In evaluations, PerfFuzz's performance proves to be more efficient than SlowFuzz's in terms of maximum path length found, for the same execution time.

3.3 Hybrid Methods

Hybrid Methods consist of methods that combine fuzzing with symbolic execution.
At first, we will talk in details about Badger [**5**], which switches between the symbolic execution engine and its fuzzer repetitively, then we will discuss other works that followed a hybrid approach and were not necessarily designed to find the worst-case complexity test cases.

- **Badger** [**5**]: Figure 4, shows how Badger combines fuzzing and symbolic execution:

  Fuzzer generates inputs and exports those that have shown increased coverage or increased *cost* on the fuzzer side to *SymExe* part. SymExe then does a concolic execution to build a partial symbolic execution tree (kept as a trie). This trie then gets expanded each time a new input is imported by SymExe. While this trie is being constructed and updated, the information about cost and coverage achieved for each node are also being gathered and updated. Using this information SymExe then chooses a most promising node (in term of maximizing cost and coverage) to start a time-bounded symbolic execution starting from that node to explore new paths and generate new inputs to export to the fuzzer. This process is repeated until a predefined execution time limit is reached. Badger's Fuzzer, *KelinciWCA* has some interesting characteristics worth mentioning. Three cost models have been defined:

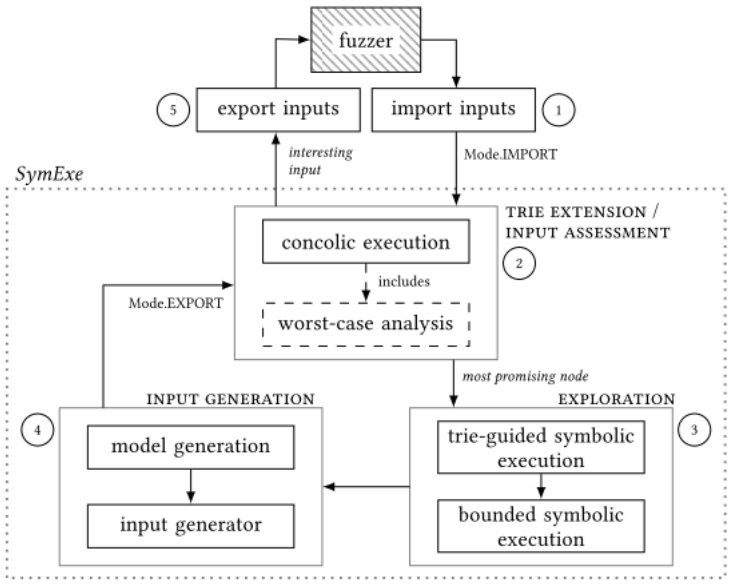


Figure 4: Badger's work flow [**5**] (figure 1 in [**5**])

1. *Timing*: Based on counting jumps (branches).
2. *Memory*: Measured by intermittent polling[3] and based on maximum consumption at any point during program's execution.
3. *User Defined*: The user can define arbitrary metrics costs to maximize a variable in the program for example.

Although Badger seems like an interesting tool, its evaluation is not promising. In most examples of the evaluation, the maximum number of jumps found by Badger, is almost the same as the maximum number found by its fuzzer.

We have also looked at other hybrid methods to find novel ideas like SAGE [**7**], Driller [**8**] and another work [**9**]. In SAGE [**7**], P. Godefroid et al., do a negation on symbolic path found by concolic execution to gain more coverage and have a better chance at finding possible bugs and vulnerabilities. The idea cannot really be adapted to our problem since it is focused on gaining more coverage. Driller has the interesting idea of running concolic execution only when the fuzzer is stuck, i.e.: "*When the fuzzing component has gone through a predetermined amount (proportional to the input length) of mutations without identifying new state transitions*". In [**10**], J. Galeotti et al., call concolic execution on an input given by the fuzzer, and then a limited symbolic execution. This individual input is chosen from the input set of fuzzer, only if after a mutation on a primitive value, a change in fitness function has occurred. Because it means that this value was important in choosing branches (in if conditions) so it is logical to call concolic and then symbolic execution on the input to derive new values.

[3]Intermittent polling is when the device is only polled at regular timed intervals.

4 Design

In this section, we will describe the design of our tool based on a symbolic execution engine, KLEE. Our tool finds the inputs to a given BPF program which trigger its worst-case execution time.

We will start by explaining our choice of method, and then continue by discussing the details of implementation, including:

- Adding BPF helpers recognition to KLEE.
- Search heuristic to find the worst-case program path.
- BPF helpers execution time evaluation.

4.1 Chosen Approach and Tool

To explain our final choice of method, we would like to remind you some of the characteristics of BPF programs:

- Limited instruction number (4096)
- Limited tail call number (32)
- Small number of BPF helpers (< 100)

Considering the limit on instruction number and permitted tail call number, we know that the maximum number of instructions to be run from executing a BPF program is $4096 * 32$ which equals to $131072 < 10^6$. As a result, we can know that if we choose to use symbolic execution we will not face path explosion. Therefore, we decided to use symbolic execution for our problem.

As said before, we chose to base our work on KLEE, a symbolic execution virtual engine.
KLEE is a symbolic virtual machine built on top of the LLVM compiler infrastructure. It takes the LLVM intermediate bytecode of a program, executes it symbolically, and generates test-cases covering most of the program paths.
KLEE is made of an interpreter loop which selects the symbolic processes (paths) to continue and furthermore execute. This interpreter loop continues its work until either all the paths are discovered and executed, or until a user-defined time limit has been reached. Process selection by this interpreter loop follows 4 heuristics:

- Depth-First Search (DFS)
- Random State Search
- Random Path Selection
- Non Uniform Random Search (NURS)

Since the solver takes a considerable amount of time to find solutions, KLEE performs a query optimization step to simplify symbolic expressions and constraints before passing them to the solver.

LLVM provides support for BPF instructions so we can get LLVM intermediate bytecode of BPF programs using tools such as Clang. Although LLVM supports BPF instructions, the source of BPF helpers isn't present in the LLVM bytecode of a BPF kernel program, since helper definitions are present in kernel and will be used once the BPF program is loaded into kernel.

Note that KLEE is a symbolic execution tool used to explore the maximum number of paths in a program; however, we are looking for the paths that take the longest time to execute. So we needed to change KLEE's exploration search mechanism. In order to find the worst-case paths, we decided to use A* heuristic which will be explained later on in more details.
To use A*, we also need to have estimations on the execution time of BPF specific instructions, and therefore, a substantial part of our work was dedicated to finding such estimations.

4.2 Adding Support for BPF Helpers

As mentioned before, KLEE is based on the LLVM compiler infrastructure, meaning that KLEE uses LLVM's internal API to interpret LLVM intermediate bytecode, its input. When BPF programs are compiled to LLVM bytecode with Clang, BPF helpers are represented differently than normal functions whose implementation is present in the program and so they are not acknowledged by KLEE. A part of our job was to add this detection and also provide support for symbolically executing BPF instructions. For the last matter, we decided not to enter into details of BPF instruction implementations and just symbolize the return value of BPF instructions. As an example, *bpf_skb_load_bytes* is a helper that loads a predefined number of bytes from a packet; our tool doesn't consider the details of how this functionality is implemented and just assumes a symbolic value for the loaded bytes of this helper.

4.3 Search Heuristic

We decided to use A* heuristic to find the most time-consuming paths in our program as previously experimented in Castan[**1**].

4.3.1 A*

A* is a graph-search heuristic algorithm which prioritizes its path exploration and it is normally used to find the least "costly" path between two given nodes in a weighted graph. It works as follows:
Imagine we want to find the most time-consuming path from a given node $S \in V$ to a node $T \in V$ in a given weighted graph $G = \{V, E\}$. Before going further, let's introduce the notion of cost function:

$$f(n) = g(n) + h(n)$$

Where n is a node in $V \in G$, and $g(n)$ is the cost of arriving from S to n and $h(n)$ is a heuristic estimating the cost of arriving at T from n.
We start by S, we look at all the neighboring nodes of S, calculate their *cost* function ($f(n)$), and put these nodes with their costs in a list of nodes left to *explore*. We also put S as a predecessor of all these nodes in order to keep the path. We will then, until we reach T or, until we have no more nodes to explore in the list, take a node with the greatest cost from the list and look at its neighbors like we did for S. If there are neighboring nodes that are present in the list of nodes to explore but with a smaller calculated cost, we will update their corresponding cost and predecessor.
A pseudo-code of A* can be found below:

Algorithm 1 A* pseudo-code

Require: $G(V, E)$, $S \in V$: Starting node, $T \in V$ Destination
Ensure: P: Longest path from S to V

```
Q ← ReversedPriorityQueue < Node, Integer >
Q.push(S, MAX_INT)
while Q.is_not_empty do
    n ← Q.pop
    n.visit ← true
    if n ≠ T then
        for all v in n.neighbors do
            if v.visit not true then
                calculate f(v)
                if Q.contains(v) & Q.get(v).value < f(v) then
                    Q.get(v).value ← f(v)
                    v.pred ← n
                else if !Q.contains(v) then
                    Q.push(v, f(v))
                    v.pred ← n
                end if
            end if
        end for
    end if
end while
```

4.3.2 Applying A* to our problem

Like in A*, by having an estimation of execution time of BPF instructions which will be explained later, we attribute a *cost* ($f(n)$) to each instruction which is equal to the sum of its *current cost* and *potential cost*. However, in contrary to most use cases of A*, we try to maximize this *cost*. The real challenge is the definition of potential cost and how it reflects whether a node is in the worst-case path or not. Let's consider $c(n_i)$, the individual cost of node n_i , as the estimation of the execution time of instruction i in the program. We can then define the current cost and potential cost of a node n_i as follows:

- **Current cost** ($g(n)$): Is the sum of arriving to n_i from S, the starting instruction's node. It is the sum of individual costs of nodes present on the path from S to n_i.
- **Potential cost** ($h(n)$): To calculate the potential cost, we consider all the paths originating from n_i to T, the terminating instruction's node and we calculate the sum of the individual costs of the nodes on each path, then choose the greatest sum as the node's potential cost.

Let's take the example from Figure 6 to understand how A* is applied with the mentioned cost definitions. On the left-hand side, we have the C code and, on the right-hand side, we have an LLVM-like bytecode corresponding to the C code.

We will at first build the control flow graph of the bytecode, shown in Figure 7. After building the flow graph, we calculate the potential cost of each instruction before starting the A* search. This potential cost is represented as c^p in the graph. To calculate the potential costs of each instruction, we start by the terminating node whose potential cost is equal to its individual instruction cost. Then, we continue by looking at its predecessors and calculate their potential cost: Potential cost of terminating node + the predecessor's individual cost. We then continue the

```
0  int foo(int a, int b)
1  {
2      b++;
3      if(b % 2 == 0)
4      {
5          b*= a;
6      }
7      b++;
8      return b;
9  }
```

Figure 5: A sample C code

Figure 6: LLVM bytecode-like from the C code in left

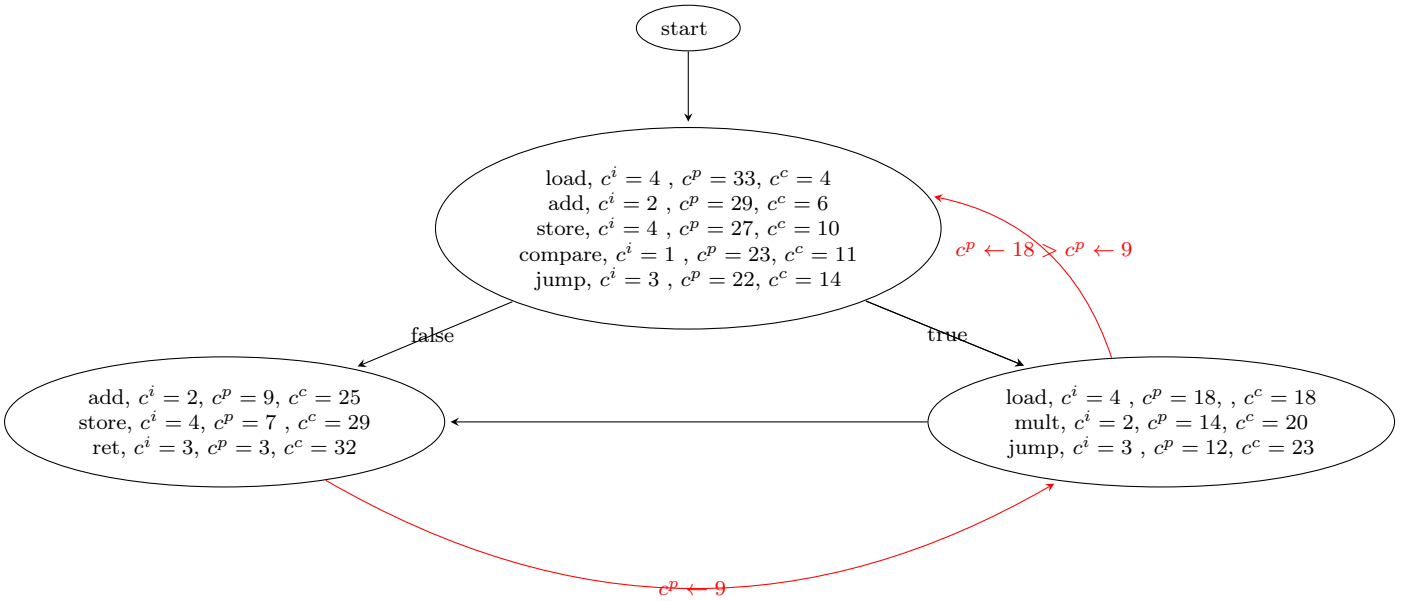


Figure 7: Flow graph with potential cost calculation path in red

same procedure recursively with the predecessors' predecessor. If we arrive to a predecessor node whose potential cost had been calculated but it was smaller than the new calculated potential cost, we update its potential cost and the potential costs of its predecessors. This procedure is a simplified *maximum flow* on the graph with the aim of maximizing potential costs. Using this method, at branches, the potential cost of jump instructions will be the maximum of the paths following this jump.
The current costs, represented as c^c are calculated at the same time A* is executed on the graph.

4.4 BPF Helpers Time Evaluation

To use A* heuristic, we need to have an estimation on the execution time of BPF helpers. At first, we started by running and measuring the execution time of the following helpers: *skb_load_bytes* and *ktime_get_ns*.
However, after trying different ways of measuring time and improving our test environment, we concluded that the execution time of *skb_load_bytes* was not stable. Since *skb_load_bytes* is a memory operation and its execution time depends on the cache state, the instability in time measurements can be due to this helper's dependence on cache state. We then, continued by measuring the execution time of independent helpers and finding out the dependencies of other helpers[4]. In order to find the different factors affecting a helpers execution time, we looked at the implementation of each helper. Each helper's execution time could be influenced by the following factors:

- **Cache state**: Cache state dependency detection is quite simple, a helper that contains a memory operation depends on the state of the cache.
- **Program's state**: A hash map lookup might take longer if the map has already been populated with many elements, in the same way, a lookup in a binary search tree with 10000 elements takes more time than a lookup in a binary search tree with 10 elements.
- **Given parameters**: A function may depend on a given parameter. For example, when calculating hash value of a key, depending on size of the key, the hash calculation time might differ.

We looked at all the basic helpers and the ones specific to *sk_filter* BPF program type, and made a table of their dependencies. This table is included in the appendix. We continued by evaluating the execution time of independent helpers by using time measurement methods discussed in the Implementation section.
For BPF instructions other than BPF helpers we haven't yet decided how to estimate their execution time. We can either take an approach like Castan's or do execution time measurements using different functions provided in the kernel.

In the following parts we discuss execution time evaluation of dependent helpers, starting with *bpf_map_lookup* in a BPF hash table map.

4.4.1 bpf_map_lookup: Hash Map

Let's recall the structure of a hash map with N elements and M buckets, where each bucket is implemented as a linked-list.

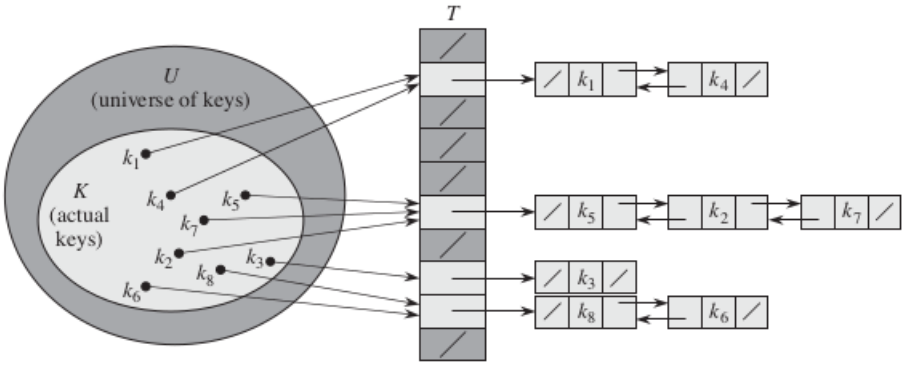


Figure 8: Hash Map, (figure 11.1 from **[17]**)

[4]This was limited to common base helpers and one program type (*sock_filter*) specific helpers.

Each element in a hash map has a key and a value. When adding new elements to a hash map, a hash function takes its key, calculates a hash value, and depending on this hash value this element will be put in a bucket m. If the hash function is perfect, i.e. the hash value of keys is uniformly distributed, the number of operations in order to lookup an element by its key will be optimal. However, if the hash function is poorly chosen, or if the table is overpopulated, we can have buckets of considerable size, meaning that looking up an element will take a lot more than just a few operations as it is expected in a near perfect hash map.

Considering our problem, the question to ask, is whether we should consider and compute the worst-case number of operations a look-up may take (N = number of elements operations)? Or should we consider the average case? Or the worst 20% of the cases ? If the probability of having the worst-case is too small then it will not be logical to take this worst-case as the basis of our evaluations.
Let's see what is the probability of having the worst-case number of operations to do a look-up, i.e., if we consider the number of operations as α, then we have $\alpha = N$. To do so, we need to consider two cases:
Either the value we are searching for is in the table or it is not.

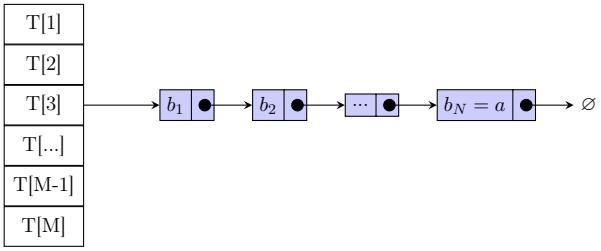


Figure 9: Worst-case lookup with the searched value, a, in the table.

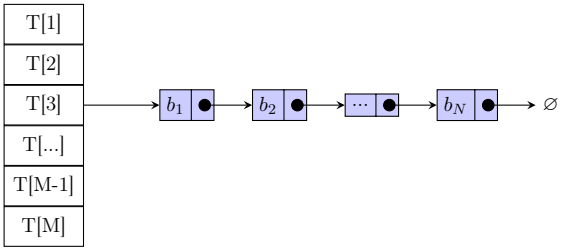


Figure 10: Worst-case lookup with the searched value not in the table.

If we consider α as the number of operations required for a lookup, and X the number of bits of the value type, the probability of the mentioned two worst cases will be:

$$P(\alpha = N) = \begin{cases} P(\alpha = N | a \notin H) = \frac{1}{M^N}(\frac{2^X - N}{2^X}) & \text{if we don't find a} \\ P(\alpha = N | a \in H) = \frac{1}{M^N}(\frac{1}{2^X}) & \text{if we find a} \end{cases}$$

The probability of having an element in a specific bucket is $\frac{1}{M}$, so the probability of having all N elements in one bucket is: $\frac{1}{M^N}$. Furthermore, in the first case, we need an a which is not equal to any of the present elements in the table. The probability of having an element from all the 2^X possible values, minus the N values corresponding to N present elements in the table. In the second case, we should be finding a equal to the last element in the bucket: $\frac{1}{2^X}$.
For realistic values of N, we can see that this probability is really small. So we decided to look at the probability of having the following cases:

$$\alpha = \{N, N-1, N-2, ..., N - 20\%N\}$$

which would be:

$$P(\alpha = N) + P(\alpha = N-1) + ...P(\alpha = N - 2 - \%N)$$

To calculate the probability of all those cases we realized that we will need to have all the possible combinations of distributing N elements in M buckets in a way that we get for example $80\%N$ elements in one bucket. Number of combinations corresponding to this one particular case is equal to:

$$\binom{80\%N+1}{M} = \frac{(80\%N+1)!}{(80\%N+1-M)!M!}$$

Calculating and keeping all such combinations for all the above cases in the memory is practically not possible. So we were obliged to change our approach. By looking at the implementation of the hash function used in BPF hash map, it looked like a near-perfect hash function. Therefore, we decided to test this hash function and see how many operations a look-up will take in average.

5 Implementation

In this part we will discuss the implementation of our tool. First, we will detail how we added support for BPF helpers, then BPF helpers' execution time evaluation.

5.1 Adding Support for BPF Helpers

BPF helpers are defined as Global Variables in LLVM bytecode as their definition is already present in the kernel. Since BPF helpers definitions are not present in the intermediate LLVM bytecode, we needed to add support for detecting these helpers, as well as, executing them symbolically. We can see the difference between BPF helpers and function declaration in an LLVM bytecode in the following figures:

```
define i32 @bpf_prog_mana(%struct.__sk_buff*) #0 section "socket1" !dbg !34 {

...
}
```

Figure 11: Function definition in LLVM intermediate bytecode

Figure 12: BPF helper ktime_get_ns in LLVM intermediate bytecode

Given a program, KLEE first finds all the function definitions in a program, and saves their addresses in a set data structure. Then, when there are function calls in the program flow, KLEE uses the set to find the functions by their address.

We followed the same approach by identifying all the BPF helpers that are defined as Global Variables in LLVM code, and saving them in a set data structure.

After this detection, we also needed to implement the symbolic execution of each BPF helper so that they will be integrated into the whole symbolic execution of the program. We decided that for our purpose, we do not have to symbolically execute the helpers in details and follow their underlying code since the output of these helpers could be treated like input variables. As an example, for a helper that loads a byte or a helper that measures time we can just attribute symbolic values to their outputs.

5.2 BPF Helpers Time Evaluation

In order to use A* heuristic in our tool, we need an estimation of the cost of BPF instructions and particularly BPF helpers. After finding helpers that did not depend on the cache state, program state or a parameter, we evaluated their execution time doing some tests. For the other helpers, we started by looking at BPF hash map's lookup operations to provide a possible worst-case execution time estimation.

5.2.1 Independent Helpers

We did execution time estimations of independent helpers using BPF_PROG_TEST_RUN command, which manually triggers a pre-specified number of runs on a pre-loaded BPF program and then returns the average execution time for one run.
We had two options to use this command, either trigger it using bpftool[5] by the following command executed in Linux shell:

[5]A tool in Linux kernel that helps manage BPF programs and maps.

```
#bpftool prog run <PROG> data_in <input> data_out <output> repeat <REP NUMBER>
```

Or, by calling *bpf_prog_test_run()*, a function that triggers BPF_PROG_TEST_RUN command with a pre-specified repetition number, in a C program executed from user space.

```
int bpf_prog_test_run(int prog_fd, int repeat, void *data,
                      __u32 size, void *data_out, __u32 *size_out,
                      __u32 *retval, __u32 *duration);
```

We continued with the second method and wrote a C program that takes the address of an executable BPF program as input, and calls *bpf_prog_test_run()* 1000 times and with the pre-specified repetition number set to 1000. The output of this C program was then written into a CSV file, treated later on by a Python script to study the standard deviation, median and mean of the found execution times.

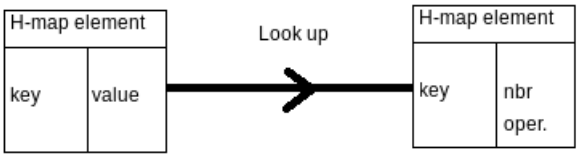
Table 1: Execution time measurements of independent helpers

	Mean (ns)	Median (ns)	std (ns)
bpf_user_rnd_u32	16.25	15.00	3.39
bpf_get_smp_processor_id	14.25	13.00	2.61
bpf_get_numa_node_id	14.11	13.00	2.45
bpf_ktime_get_ns	37.17	34.00	6.24

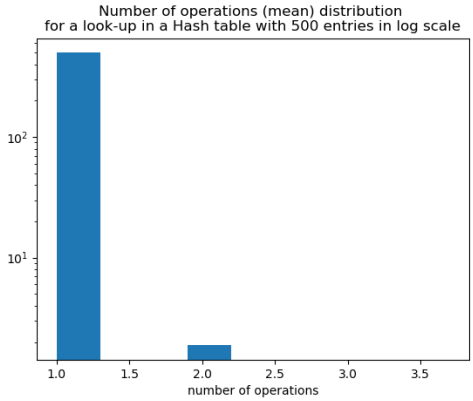
You can find the corresponding box-plots with outliers in the appendix 6.

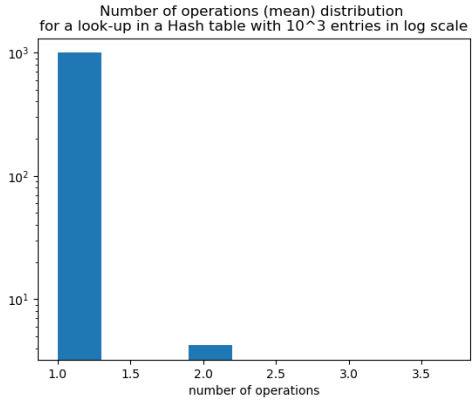
5.2.2 bpf_map_lookup: Hash map

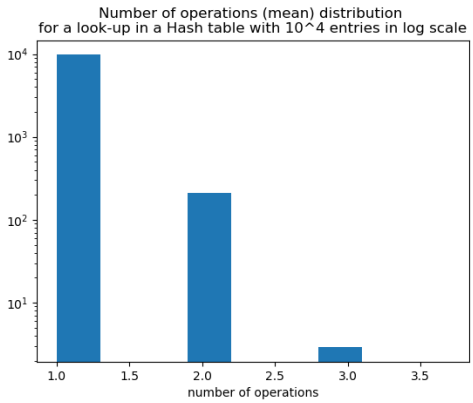
As mentioned in the Design section, we decided to test this hash function by measuring how many operations a lookup will take in average. More precisely, we look at the number of operations performed when looking in a bucket list. To do so, we changed the lookup operation in BPF hash map's implementation in the kernel, by counting the number of operations and replacing the value part in the map element by this counted number, and recompiled the kernel. It means that after looking up a key, we then had the number of operations it took to do the lookup as the hash map element's value:

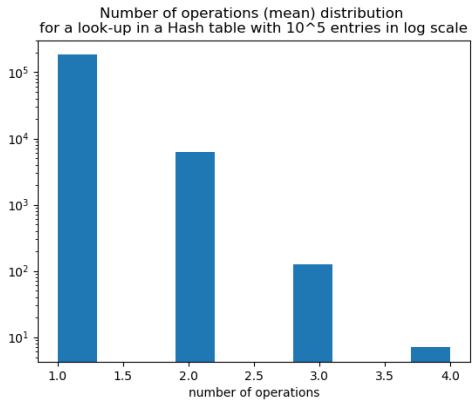


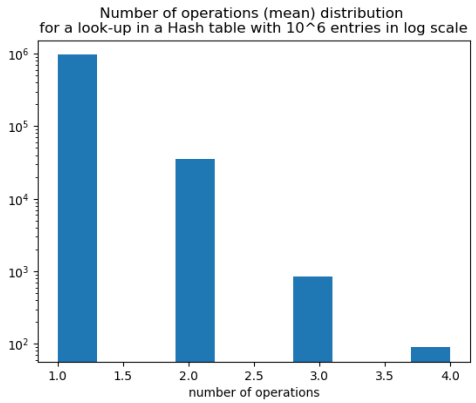
In this modified kernel, we then, initialized BPF hash maps of different initial sizes and populated them with elements of distinct random (uniform seed) keys. We looked up these keys to have the number of operations it took to find those keys in map and visualized them in a histogram with logarithmic scale.











We can see the actual corresponding numbers in the following table:

Table 2: Mean of Number of operations for a lookup in a hash table with $[500 - 10^6]$ entries

	500	10^3	10^4	10^5	10^6
1	498.5	995.77	9787.9	183585.3	963281.2
2	1.875	4.22	209.5	6288.7	35855.8
3	0	0	2.889	125.9	845.1
4	0	0	0	7	89.5

6 Conclusion

The aim of this project was to design a tool that takes a BPF program and generates the inputs to this BPF program that trigger its worst-case execution time and also to know what this worst-case execution time is. Throughout this internship we studied different software analyzing approaches and the related works in order to find the method that best suited our problem, in particular, considering its specific characteristics such as, the limit on the number of instructions. We decided to take symbolic execution as the approach to solve our problem and generate worst-case execution time test-cases. In order to find the most time-consuming path in the flow graph, we chose A* as heuristic, which introduced its own challenge: definition of potential cost for each instruction. This potential cost was eventually defined as the worst cost of all the instructions paths following an instruction. Furthermore, we needed to have an estimation on the cost of each BPF helper and symbolize its behavior. To do the former, we had to study the factors affecting the execution time of each BPF helper and treat them accordingly.
In this work, we have not considered accessibility to BPF maps from outside the BPF programs. Memory interactions are not integrated into the tool neither and we do not consider the realistic cost of specific memory operations, such as cache misses, on the worst-case execution time.

At the moment, we need to estimate execution time of the rest of the helpers and add the A* heuristic to our tool which already detects BPF helpers, and also implement symbolic behavior of the remaining BPF helpers. We can then, proceed by evaluating our tool by giving it different BPF programs with known worst-case algorithmic complexity and compare the worst case found by our tool with the theoretical worst case.

Appendix

BPF base helpers:

- BPF_FUNC_map_lookup_elem
- BPF_FUNC_map_update_elem
- BPF_FUNC_map_delete_elem
- BPF_FUNC_map_push_elem
- BPF_FUNC_map_pop_elem
- BPF_FUNC_map_peek_elem
- BPF_FUNC_get_prandom_u32: Gets a pseudo-random number.
- BPF_FUNC_get_smp_processor_id: Gets the symmetric multiprocessing processor id.
- BPF_FUNC_get_numa_node_id: Returns the id of the current NUMA node
- BPF_FUNC_tail_call: Provides a tail call to another BPF program.
- BPF_FUNC_ktime_get_ns: Returns the time elapsed since system boot, in nanoseconds.

The list of all helpers corresponding to a program type could be found by typing the following command in Linux kernel's root:

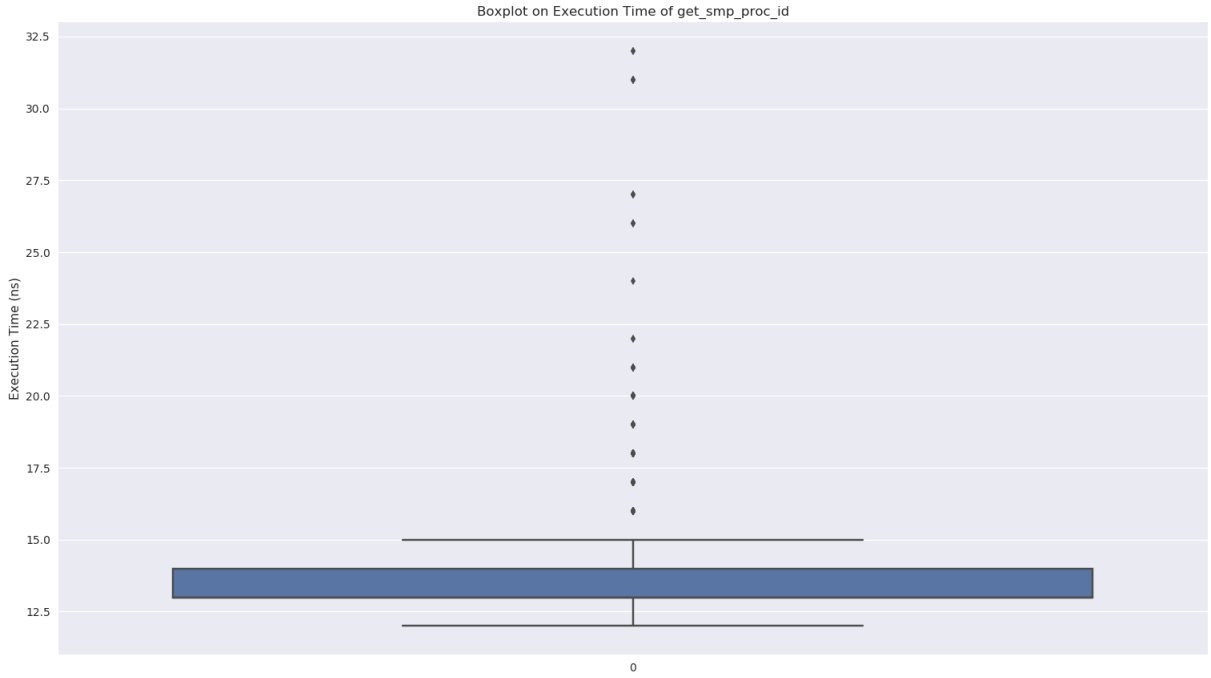
```
git grep -W 'func_proto(enum bpf_func_id func_id' kernel/ net/ drivers/
```

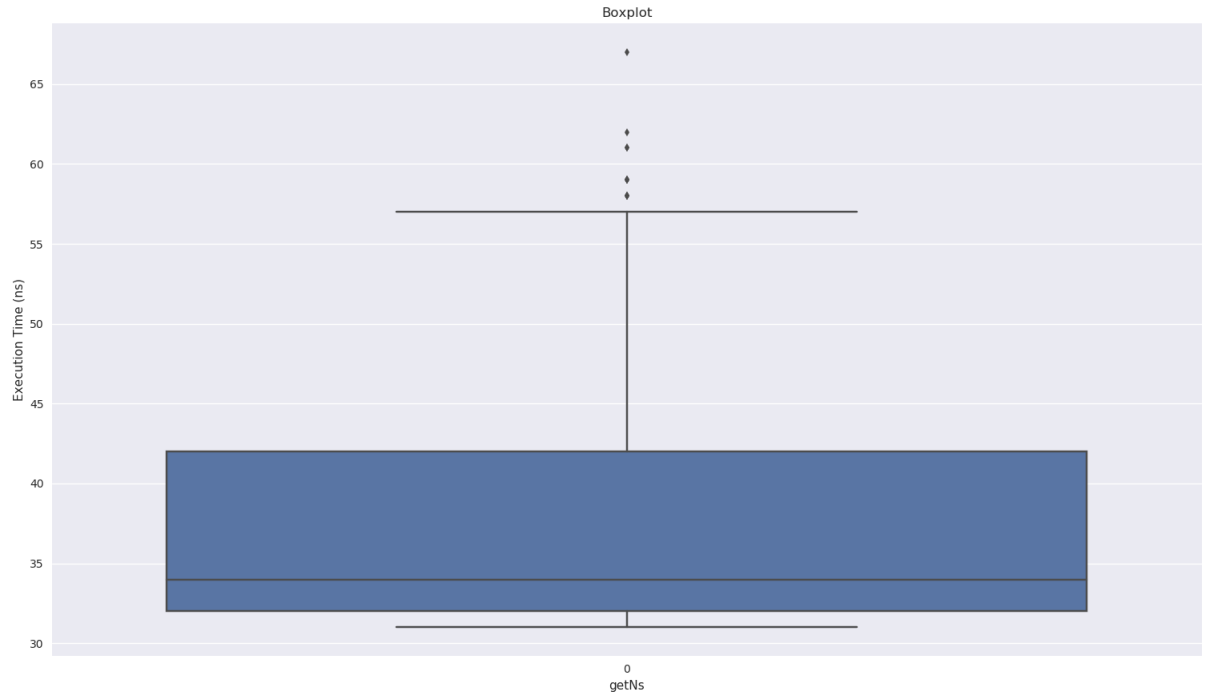
Basic Helpers	Cache	Parameter	State
bpf_map_lookup:			
PERCPU_HASH	x	x	x
LRU_PERCPU_HASH	x	x	x
PERCPU_ARRAY	x		
PERCPU_CGROUP_STORAGE	x		x
CGROUP_STORAGE	x	.	
BPF_MAP_TYPE_STACK_TRACE	x	x	
PROG_ARRAY	x		
PERF_EVENT	x		
CGROUP_ARRAY	x		
ARRAY_OF_MAPS	x		
HASH_OF_MAPS	x	x	x
REUSEPORT_SOCKARRAY	x		
QUEUE	x		
STACK	x		
HASH	x	x	x
ARRAY	x	.	
LPM_TRIE	x		x
DEVMAP	x		
SOCK_MAP	x		
SK_STORAGE	x		x
SOCK_HASH	x	x	x
XSK_MAP	x		
CPU_MAP	x		
bpf_map_update_elem:			
LRU_PER_CPU_HASH	x	x	x
PER_CPU_HASH	x	x	
PERCPU_CGROUP_STORAGE	x	x	
CGROUP_STORAGE	x	.	
PER_CPU_ARRAY	x	x	
PROG_ARRAY	x		
PERF_EVENT	x		
CGROUP_ARRAY	x		
ARRAY_OF_MAPS	x		
HASH_OF_MAPS	x	x	x
REUSEPORT_SOCKARRAY	x		
QUEUE	x	.	
STACK	x	.	
HASH	x	x	x
ARRAY	x	.	
LPM_TRIE	x	.	x
DEVMAP	x	.	.
SOCK_MAP	x		
SK_STORAGE	x	.	x
SOCK_HASH	x	x	x
XSK_MAP	x	.	
CPU_MAP	x	.	x
bpf_map_delete_elem:			
BPF_MAP_TYPE_LRU_PER_CPU_HASH	x	x	x
BPF_MAP_TYPE_LRU_HASH	x	x	x
BPF_MAP_TYPE_PER_CPU_HASH	x	x	x
BPF_MAP_TYPE_PER_CPU_CGROUP_STORAGE			
BPF_MAP_TYPE_CGROUP_STORAGE			
BPF_MAP_TYPE_PER_CPU_ARRAY			
PROG_ARRAY	x		
PERF_EVENT	x		
CGROUP_ARRAY	x		
ARRAY_OF_MAPS	x		
HASH_OF_MAPS	x	x	x
REUSEPORT_SOCKARRAY	x		
QUEUE			
STACK			
BPF_MAP_TYPE_HASH	x	x	x
ARRAY			
LPM_TRIE	x		x
DEVMAP	x		
SOCK_MAP	x		
SK_STORAGE	x		x
SOCK_HASH	x	x	x
XSK_MAP	x		
CPU_MAP	x		x
bpf_user_rnd_u32			
bpf_get_smp_processor_id			
bpf_get_numa_node_id			
bpf_tail_call			
bpf_ktime_get_ns			
bpf_trace_printk	x	x	

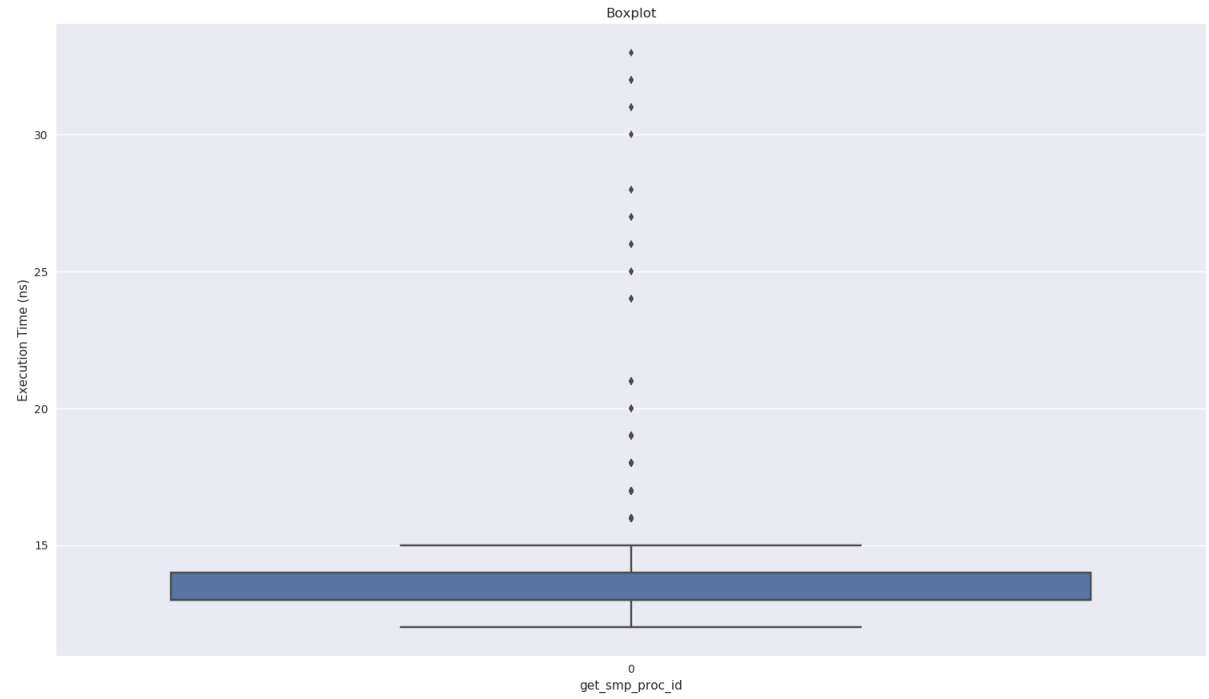
sock_filter	Cache	Parameter	State
bpf_get_current_uid_gid	x		
bpf_get_local_storage	x		

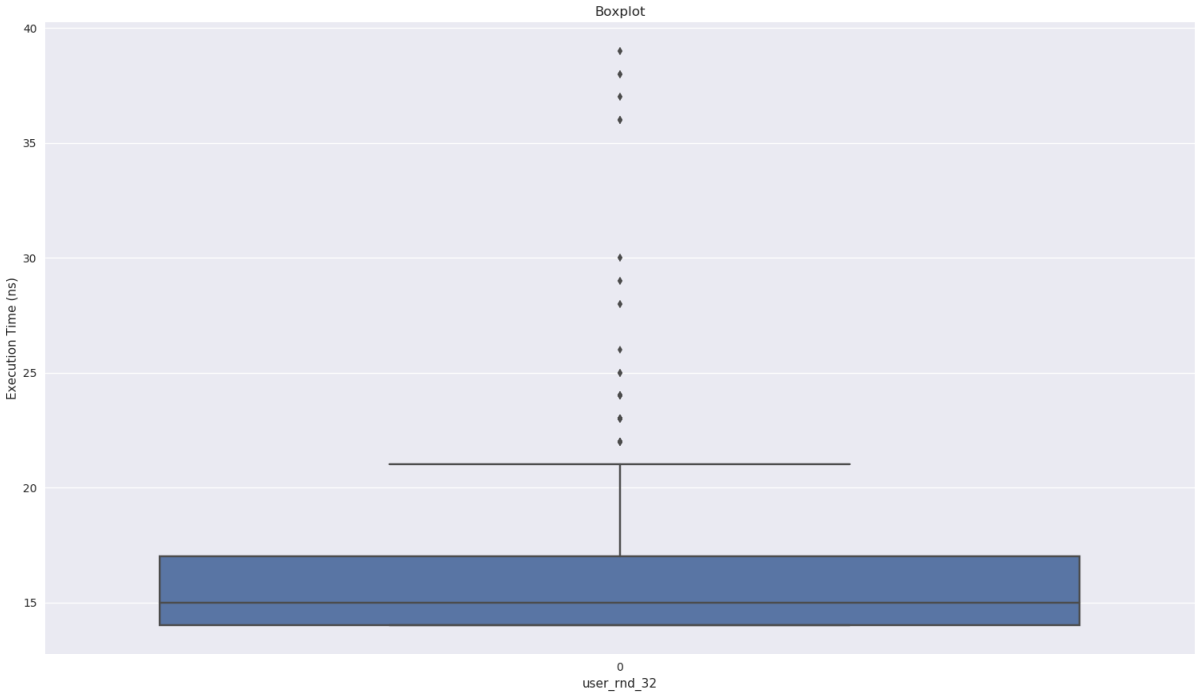
sk_filter	Cache	Parameter	State
bpf_get_socket_cookie	x		
bpf_get_sock_id	x		

Independent helpers execution time measurement:






Boxplot
Execution Time (ns)
30
25
20
15
0
get_smp_proc_id


Boxplot
Execution Time (ns)
40
35
30
25
20
15
0
user_rnd_32

References

[1] Luis Pedrosa, Rishabh Iyer, and Arseniy Zaostrovnykh. *Automated Synthesis of Adversarial Workloads for Network Functions.* Sigcomm '18, 2018.

[2] Burnim, Jacob Juvekar, Sudeep Sen, Koushik *WISE: Automated Test Generation for Worst-Case Complexity.*

International Conference on Software Engineering (ICSE), 2009.

[3] Petsios, Theofilos Zhao, Jason Keromytis, Angelos D. Jana, Suman *SlowFuzz: Automated Domain-Independent Detection of Algorithmic Complexity Vulnerabilities.* ??, ??.

[4] Lemieux, Caroline Padhye, Rohan Sen, Koushik Song, Dawn *PerfFuzz: automatically generating pathological inputs.* ??, 2018.

[5] Noller, Yannic Kersten, Rody Păsăreanu, Corina S. *Badger: Complexity Analysis with Fuzzing and Symbolic Execution.* ??, 2018.

[6] Wikipedia: Fuzzing,
`https://en.wikipedia.org/wiki/Fuzzing`

[7] Godefroid, Patrice Levin, Michael Y. Molnar, David *SAGE: Whitebox Fuzzing for Security Testing.* Queue, 2012.

[8] Stephens, Nick Grosen, John Salls, Christopher Dutcher, Andrew Wang, Ruoyu Corbetta, Jacopo Shoshitaishvili, Yan Kruegel, Christopher Vigna, Giovanni *Driller: Augmenting Fuzzing Through Selective Symbolic Execution.* , 2017.

[9] Cadar, Cristian Dunbar, Daniel Engler, Dawson *KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs.* OSDI'08 Proceedings of the 8th USENIX conference on Operating systems design and implementation , 2008,

[10] Galeotti, Juan Pablo Fraser, Gordon Arcuri, Andrea *Improving search-based test suite generation with dynamic symbolic execution.* 2013 IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, 2013.

[11] Galeotti, Juan Pablo Fraser, Gordon Arcuri, Andrea *Improving search-based test suite generation with dynamic symbolic execution.* 2013 IEEE 24th International Symposium on Software Reliability Engineering, ISSRE 2013, 2013.

[12] `https://cilium.readthedocs.io/en/latest/bpf/#maps`

[13] `https://github.com/klee/klee`

[14] `https://llvm.org/doxygen/classllvm_1_1GlobalVariable.html`

[15] Oechslin, Philippe *Making a Faster Cryptanalytic Time-MemoryTrade-Off.* Annual International Cryptology Conference,2003.

[16] `https://en.wikipedia.org/wiki/Just-in-time_compilation`

[17] Corman, Thomas Leiserson, Charles Rivest, Ronald Stein, Clifford *Introduction to Algorithms.*