Fuzzing the Berkeley Packet Filter

By

Benjamin Curt Nilsen

Thesis

Submitted in partial satisfaction of the requirements for the degree of

Masters

in

Computer Science

in the

Office of Graduate Studies

of the

University of California

Davis

Approved:

Professor Hao Chen, Chair

Professor Cindy Rubio González

Professor Matt Bishop

Committee in Charge

June 2020

Contents

List of Figures

List of Tables

Abstract

Fuzzing the Berkeley Packet Filter

The Berkeley Packet Filter (BPF) allows users to write source code that can run in kernel space. The original intent of BPF was to allow users to filter incoming network packets at the kernel level, preventing the overhead cost of switching between user and kernel mode for filtering activities. Now BPF (also known as extended BPF) allows users to write BPF type programs in C to run in the kernel mode. These BPF programs utilize various datatypes and helper methods and can be compiled with a gcc type compiler. Companies such as Facebook are starting to use this extended BPF functionality which makes it a great target for a thorough project. Because these BPF programs run in kernel space, it is imperative that they be thoroughly tested for software vulnerabilities. I present a set of unique driver programs and program generators to help us run existing fuzzing tools on the BPF environment. Later I discuss the results and expand on future work and present ideas on how to improve fuzzers.

Acknowledgments

I would like to thank my advisor, Professor Hao Chen; my Thesis Committee of Professor Matt Bishop and Professor Cindy Rubio González; and Sandia National Labs for giving me the opportunity to work on this research. I learned many valuable new things and am grateful for their support.

Chapter 1

INTRODUCTION

Traditional fuzzing involves very little work for the developer. Download and compile the program binary, run the fuzzer command and wait for results. However the BPF ecosystem is more of a collection of tools for the user to create their own 'BPF' program. At the new high level (extended BPF), users are now able to use a BPF Application Programming Interface (API) library to create their own BPF maps and BPF programs. This raises the issue of how to ensure a complete fuzzing expedition has been performed over a library of dependent functions?

Fuzzing has been a powerful software technique to uncover hidden bugs in a program. This technique does not require any insider knowledge about the target program being fuzzed. Other ways of bug discovery often involve having a high level of expertise with the inner workings of the program. Formal methods is a hot area of research as it focuses on mathematical techniques for specification and proofs during the development stage of software and hardware systems. This specification inherently requires a high level of knowledge with the program, making it available to only a select number of programs/ developers. Recent state of the art research suggests grey-box fuzzers are great at finding real world bugs while requiring no insider information on the program.

The Berkeley Packet Filter (BPF) is a technology available in Unix-like operating systems to allow userspace programs to run entirely in the kernel, saving valuable time. The traditional BPF was designed to solely filter packets in the kernel (hence the name packet-filter), however today the BPF often refers to eBPF or extended BPF. The ex-

tended BPF allows users to write any program - not just a packet filter - to run in kernel space. A senior performance architect at Netflix describes BPF as giving a program "observable super powers". Facebook engineers use BPF as part of a network load balancer. According to the senior engineer "BPF is now the name of a technology and not just an acronym ... BPF is the biggest operating systems change I've seen in my career". Both these remarks were from an article posted recently (Dec 2019) [11]. Many well-known tech companies are starting to adopt this BPF technology, and due to its big change in operating systems performance and reliance on running in the volatile Linux-kernel, my advisor and I thought it would be a great target to explore for vulnerabilities.

With a target such as BPF, we must begin by discussing ways to adequately explore bugs. As discussed prior, formal methods require work on the developer side, making it inadequate for this project. We would need a tool that has a great track record of exploring source code and unveiling bugs while requiring no knowledge of how the program was built. Therefore, I decided to go with greybox fuzzing, as I had access to the source code without any developer knowledge of its design. To achieve the best results, I decided to use Angora [7]: a state-of-the-art fuzzer . For more analysis, I included the more traditional and well-known fuzzer AFL [8].

Chapter 2

BACKGROUND

The Berkeley Packet Filter is a special-purpose virtual machine for filtering network packets. The extended BPF (eBPF) contains an instruction set used to construct a BPF program. This is a RISC-type instruction set allowing these programs to be written in C and compiled into BPF instructions to run in the kernel. The BPF Programs can still talk to processes running in user space via a BPF Map. These maps are structures created and defined by the user. This BPF language contains 11 64-bit registers with 32-bit subregisters and a 512 byte stack space. A BPF program can call a predefined helper function which is defined by the core kernel. [2]

The BPF calling convention is as follows: 10 registers

r0: contains the return value of the helper function call.
r1 - r5: hold arguments from the BPF program to the kernel helper function.
r6 - r9: are callee saved registers that will be preserved on helper function call.

The maximum number of BPF instructions per program is restricted to 4096, ensuring programs terminate quickly.

There are many advantages to running these instructions in the kernel, such as optimizing performance by compiling out features the program doesn't use, and providing a stable ABI towards user space that does not require a third party kernel module [3].

The Linux kernel is shipped with a BPF interpreter which executes programs assem-

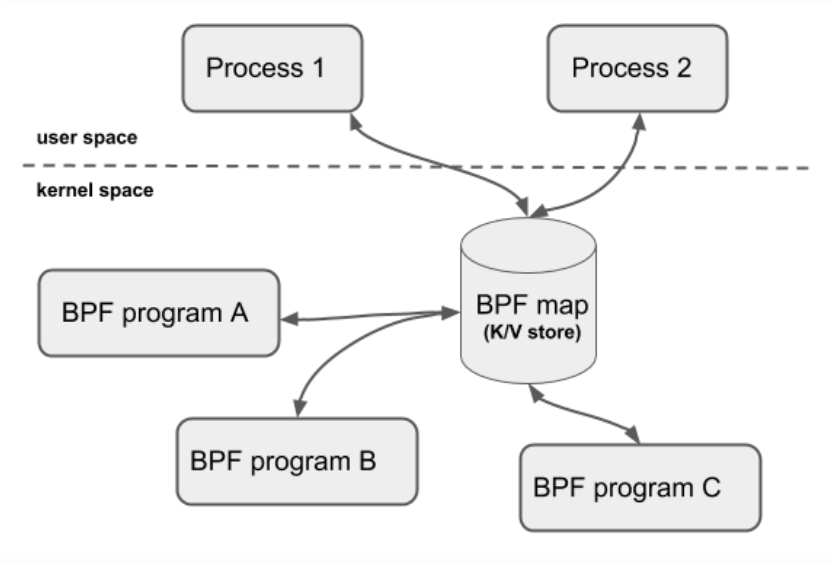

Figure 2.1. A Diagram of the BPF Map structure.

bled in BPF instructions. All BPF handling such as loading programs into the kernel of the creation of BPF maps is managed through a central bpf() system call.

BPF Maps

BPF maps are efficient key / value stores that can reside in kernel space. These maps are responsible for the cross communication between the user space and the BPF programs residing in kernel space. The BPF Map types can be seen above in table 2.1 [5]. Programs of different BPF types can share the same BPF map.

2.1 PREVIOUS WORK

Some researchers have recently created a test fuzzing environment for loading BPF programs with various mutated instructions [1]. This project has been a work of progress since 2015 and has claimed to have found one bug. After a review of their source code, I was able to find a few areas with room for improvement. First, they depend on libFuzzer as their fuzzer. More recent state-of-the-art fuzzers could provide a more thorough fuzzing

Table 2.1. BPF Map Types

BPF Map Types
BPF_MAP_TYPE_UNSPEC
BPF_MAP_TYPE_HASH
BPF_MAP_TYPE_ARRAY
BPF_MAP_TYPE_PROG_ARRAY
BPF_MAP_TYPE_PERF_EVENT_ARRAY
BPF_MAP_TYPE_PERCPU_HASH
BPF_MAP_TYPE_PERCPU_ARRAY
BPF_MAP_TYPE_STACK_TRACE
BPF_MAP_TYPE_CGROUP_ARRAY
BPF_MAP_TYPE_LRU_HASH
BPF_MAP_TYPE_LRU_PERCPU_HASH
BPF_MAP_TYPE_LPM_TRIE
BPF_MAP_TYPE_ARRAY_OF_MAPS
BPF_MAP_TYPE_HASH_OF_MAPS
BPF_MAP_TYPE_DEVMAP
BPF_MAP_TYPE_SOCKMAP
BPF_MAP_TYPE_CPUMAP
BPF_MAP_TYPE_XSKMAP
BPF_MAP_TYPE_SOCKHASH
BPF_MAP_TYPE_CGROUP_STORAGE
BPF_MAP_TYPE_REUSEPORT_SOCKARRAY
BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE
BPF_MAP_TYPE_QUEUE
BPF_MAP_TYPE_STACK
BPF_MAP_TYPE_SK_STORAGE
BPF_MAP_TYPE_DEVMAP_HASH

```
struct bpf_test {
   const char *descr;
   struct bpf_insn insns[MAX_INSNS];
   int fixup[32];
   const char *errstr;
   enum {
      ACCEPT,
      REJECT
   } result;
   enum bpf_prog_type prog_type;
};
```

Figure 2.2. BPF Test Code

experience. This section will talk about their method and discuss possible room for improvement.

Their bpf_test struct in figure 2.2 has the max insns array set to a max length of 512. For this project, we rely on the principle that many programs crash when given inputs not intended by the developer. In this case, we allow for the possibility of well over 512 instructions to be loaded in with a libbpf method to thoroughly test the error handling abilities. The driver code for their fuzzer utilizes a create_map() function which only utilizes one of the BPF_MAP_TYPES (BPF_MAP_TYPE_HASH).

The test_verifier.c class has multiple pre-made bpf instructions to test on the BPF program load (bpf_prog_load) method. We propose a new way of auto-generating bpf instructions to load into the bpf_prog_load(). Traditionally the bpf_prog_load() methods take in arguments indicating the size of the instructions sizeof(bpf_insn), however we will also test these arguments with randomly generated values from the fuzzer to detect any possible bugs with error handling.

Next we discuss the possibility of method ordering. The previous attempts of testing BPF have a simple ordering of the way they call bpf methods. For example, at the end

of all their fuzzing runs, they have a bpf_free_map() command. We propose a newer technique of letting the fuzzer determine the order of the methods being called. This way we can test possible bpf_map_free() commands before the bpf_map has even been created, look for elements that haven't been stored, etc. To do this, we design our system with the following principle in mind: Allow the fuzzer to often create syntactically correct inputs and method orderings while allowing it to produce "invalid" bpf instructions and method orderings to thoroughly test all possible ways the BPF can be used.

Lastly, we will be allowing the fuzzer to determine the number of times these methods are called. This helps us catch the potential bug with memory errors from the build-up of bpf object creation.

2.1.1 Challenges

This thesis will mostly focus on the creation of BPF programs and the libbpf API. A basic program is trivial to fuzz, but complex libraries of methods that need to be constructed on an individual basis become much harder. We hope this work of exploring the ways of fuzzing BPF can be applied to other similar program structures as well. The following questions will be discussed: How do we ensure a complete BPF fuzzing experience has been performed? How can we design a system to properly mutate BPF instructions to adequately explore and trigger all possible bpf paths? How can we ensure our project is adequately exploring the libbpf source code? How can we apply the concepts discussed above to our fuzzing expedition?

The example below in figure 2.3 shows how a BPF program can be offloaded to the Netronome Flow Processor (NFP) using libbpf calls [2]. For driver mode, ifindex should be set to 0, for offload it should be set to the NFP interface index.

We can see this code takes in a specific file (file.o) and uses a single BPF program type. All traditional BPF Programs are not designed with a fuzzer in mind.

Below in figure 2.3 is an example program with a BPF Program Load command with a given set of BPF instructions.

```c
#include <linux/bpf.h>
#include <linux/if_link.h>
#include "bpf/libbpf.h"

int main(void)
{
   struct bpf_prog_load_attr prog_load_attr = {
      .prog_type = BPF_PROG_TYPE_XDP;
   };
   int prog_fd;
   static int ifindex;
   static __u32 xdp_flags;
   struct bpf_object *obj;
   ifindex = 3;
   prog_load_attr.file = "file.o";
   prog_load_attr.ifindex = ifindex; /* set offload dev ifindex */
   xdp_flags |= XDP_FLAGS_HW_MODE; /* set HW offload flag */

   if (bpf_prog_load_xattr(&prog_load_attr, &obj, &prog_fd))
      return 1;
   if(!prog_fd){
      printf("error loading file");
   }
   if(bpf_set_link_xdp_fd(ifindex, prog_fd, xdp_flags) < 0) {
      printf("link set xdp fd failed\n");
      return 1;
   }
   return 0;
}
```

Figure 2.3. BPF Program Example

The code above demonstrates typical BPF programs that utilize libbpf functions and BPF instructions. The challenge presents itself: how do we construct a program capable of feeding in valid inputs to a libbpf method?

2.2 Design

I chose Angora [6] to be the main fuzzer behind this project as it has proven itself to be the most successful C fuzzer to date. Angora has implemented a few state-of-the-art techniques that can be utilized to create our driver program.

2.2.1 Scalable Byte Level Tracking

The first one of these techniques is scalable byte level tracking. Many path constraints only depend on a few bytes. Angora keeps tracks of these bytes, and mutates them accordingly. Other fuzzers (AFL) mutate the entire input. To fuzz the wide array of Berkeley Packet Filter methods, I need to allow the fuzzer to choose which methods are called. The mutated files generated from Angora will contain a series of bits that are used to fill a struct with data.

This "pta" struct we created contains booleans m1 - mn to dictate the calling of a method. The driver program reads in a binary file as the argument (from the fuzzer), and fills this binary data into the pta struct. Because of byte level tracking, Angora can properly mutate these files to choose which methods get taken. Angora's other state-of-the-art feature includes shape and type inference.

2.2.2 Context Sensitive Branch Coverage

Context Sensitive Branch Coverage will be utilized in a similar way as scalable byte level tracking. In the case of the Berkeley Packet filter, the context is very important when navigating branches. Methods can be called in any order that load programs, create maps, add elements to maps, delete elements from maps, etc. It is sensible to know the context when calling a "delete element" method, and Angora helps us keep track of these

```
static int test_sock(void)
{
   int sock = -1, map_fd, prog_fd, i, key;
   long long value = 0. tcp_cnt, udp_cnt, icmp_cnt;
   map_fd = bpf_create_map(BPF_MAP_TYPE_ARRAY, sizeof(key), sizeof(value),
       256, 0);
   if (map_fd < 0) {
      printf("failed to create map '%s'\n", strerror(errno));
      goto cleanup;
   }
   struct bpf_insn prog[] = {
      BPF_MOV64_REG(BPF_REG_6, BPF_REG_1),
      BPF_LD_ABS(BPF_B, ETH_HLEN + offsetof(struct iphdr, protocol) /*R0 =
          ip->proto */),
      BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_0, -4), /* *(u32 *)(fp - 4) =
          r0 */
      .
      .
      BPF_MOV64_IMM(BPF_REG_1, 1), /* r1 = 1 */
      BPF_RAW_INSN(BPF_STX | BPF_XADD | BPF_DW, BPF_REG_0, BPF_REG_1, 0, 0),
          /* xadd r0 += r1 */
      BPF_MOV64_IMM(BPF_REG_0, 0), /* r0 = 0 */
      BPF_EXIT_INSN(),
   };
size_t insns_cnt = sizeof(prog) / sizeof(struct bpf_insn);
prog_fd = bpf_load_program(BPF_PROG_TYPE_SOCKET_FILTER, prog, insns_cnt,
    "GPL", 0, bpf_log_buf, BPF_LOG_BUF_SIZE);
}
```

Figure 2.4. BPF Program Example

```
typedef struct args{
   int one;
   int two;
   bool m1;
   //etc....
} args_t;
args_t pta;
```

Figure 2.5. PTA struct

```
if(pta.m2){
   printf("populating prog array\n");
   populate_prog_array(&pta.event, pta.prog_fd);
}

if(pta.m3){
   printf("Loading and attaching\n");
   load_and_attach(&pta.event2, &pta.prog, pta.size);
}

if(pta.m4){
   printf("Creating a BPF map\n");
   bpf_create_map(pta.map_type, pta.key_size, pta.value_size, pta.max_entries,
      pta.map_flags);
}
```

Figure 2.6. Method Selection

```
typedef struct args{
   int random;
   struct bpf_map_data maps;
   int nr_maps;
   .
   .
   bool m3;
} args_t;
args_t pta;
FILE *fp;
fp = fopen(argv[1], "rb");
if(fread(&pta, sizeof(pta), 1, fp) != 1){
   printf("File not big enough\n");
   fclose(fp);
   return 0;
}
```

Figure 2.7. File Input

important details. With this and the combination of Scalable Byte level tracking, we can create a driver program that calls the program in a random (but deterministic) order while keeping track of the context when doing so.

2.2.3 Shape and Type Inference

As noted earlier, our driver program reads in a single file from Angora. We take this binary file and fill in a struct with values. See implementation in figure 2.7. Because of Angora's ability to infer the shape and type of the values, it can fuzz more efficiently.

To get a better understanding of Angora's path coverage metrics, I created a series of custom path test classes. These I designed with a series of method calls and if() statements, so that I could learn what triggers a new path within Angora. This also gave

```
int bpf_prog_get_fd_by_id(__u32 id)
{
   union bpf_attr attr;
   memset(&attr, 0, sizeof(attr));
   attr.prog_id = id;
   //ADDING INJECTED BUG:
   __u32 id2 = 4;
   if(id == id2){
      raise(SIGSEGV); //INJECTED BUG TRIGGERED
   }
   return sys_bpf(BPF_PROG_GET_FD_BY_ID, &attr, sizeof(attr));
}
```

Figure 2.8. Injected Bug

me a baseline pathcount for my default driver program, so that I would not count these paths explored in my driver program as paths of BPF code covered.

To ensure the driver program was capable of finding bugs, I created a few test cases with injected bugs. These bugs were hidden behind nested conditional if-statements, and Angora was able to locate all injected bugs. This confirmed my driver program was filling my argument struct correctly and quickly mutating values to pass the conditions required by the test program to trigger the bug.

An example of simple injected bug can be found in figure 2.8 - found instantly

Chapter 3

New Fuzzing Techniques

To properly fuzz this BPF ecosystem, we present 3 new techniques for applying state of the art path-guided fuzzers on libbpf. First, we present a code-generation based fuzzing technique capable of generating hundreds of instrumented and libbpf filled binaries for thorough testing. Second, we create an individual class capable of running all possible combinations of libbpf method calls. Third, we leverage the strengths of Angora and other path-based fuzzers to generate a valid set of BPF instructions to load into the kernel.

3.1 Program Generation

Aside from the previously mentioned BPF fuzzing tool, the only other way of fuzzing BPF code involved fuzzing existing BPF Tool example files found in the Linux kernel. Unfortunately, there was very little progress made by fuzzing these existing tools. With only 20 odd small classes to fuzz, I decided more BPF classes were needed with a wide variety of method calls to fuzz. I designed a tool to create various BPF C programs to achieve more code coverage the previous classes may not have exposed. Our first technique relies on a C++ program I engineered capable of generating various C classes containing libbpf methods. These classes all contain our standard Angora-driver code capable of reading in files and generating valid arguments for these BPF methods. This is followed by generated libbpf method calls to ensure all possible orders of method calls have been produced and fuzzed.

Below in figure 3.1 is a diagram of our BPF Generator Model.

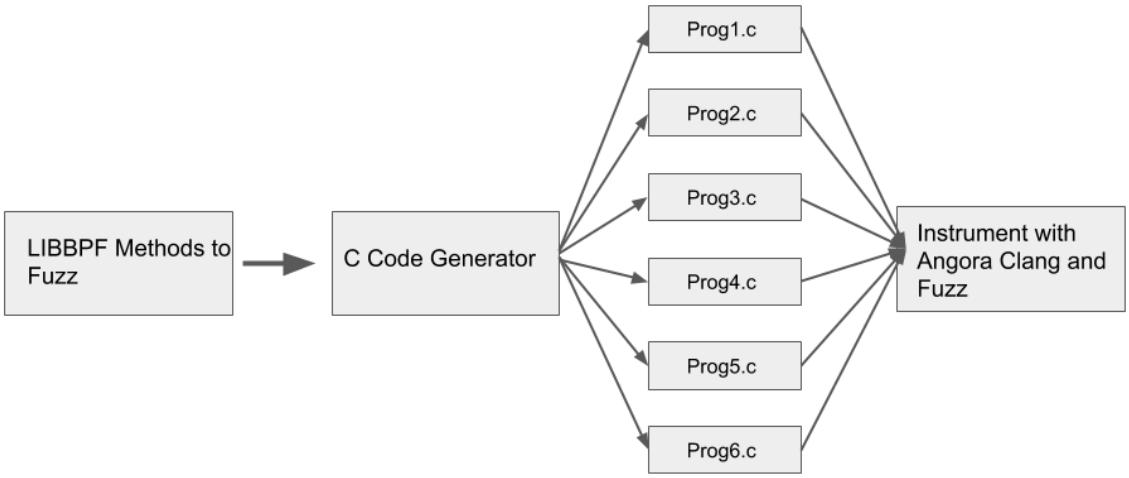

Figure 3.1. A diagram of the BPF Program Generator.

After a series of files has been created containing various BPF methods, my program generates a Makefile to compile all these programs with Angora. I made an AFL version as well. Then I created a script that can launch my program generating program, and run the corresponding generated makefile to create hundreds of instrumented binaries for us to fuzz.

This new form of testing enabled us to find a few bugs, however there are some drawbacks to this approach. The biggest one being the creation of too many classes, since each class created needs to be fuzzed for roughly 48 hours on an Intel Xeon CPU E7-4850 @ 2.00GHz. Therefore if someone wanted to fuzz all generated classes, it would take a considerable amount of time. This leads us to our next technique of single class - multiple method call class fuzzing.

```
apicall[8] = "bpf_program__set_socket_filter(&bo);\n";
apicall[9] = "bpf_program__set_tracepoint(&bo);\n";
.
.
apicall[23] = "bpf_program__is_xdp(&bo);\n";
apicall[24] = "bpf_program__is_perf_event(&bo);\n\n";

int filenum = 0;
string firstAPI;
for(int i = 0; i < 25; i++){
   code = standardTemplate;
   code += apicall[i];
   firstAPI = code;
   for(int j = 0; j < 25; j++){
      code += apicall[j];
      code += "\nreturn 0;\n}";
      num = to_string(filenum);
      filename +=num;
      filename += dotc;
      fp[filenum].open(filename);
      fp[filenum] << code << endl;
      code = firstAPI;
      filenum++;
      filename = fileNS;
      fp[filenum].close();
   }
}
```

Figure 3.2. BPF API Calls

3.2 Variable Method Call Fuzzing

The drawback of having to fuzz multiple files lead us to the creation of a single class capable of fuzzing all possible combinations of BPF calls. The class contains a similar file input system as described earlier, however there is additional logic allowing the fuzzer to give inputs which determine the number and order of called methods. One of the biggest problems with fuzzing the entire libbpf system is the inherent complexity of the entire design.

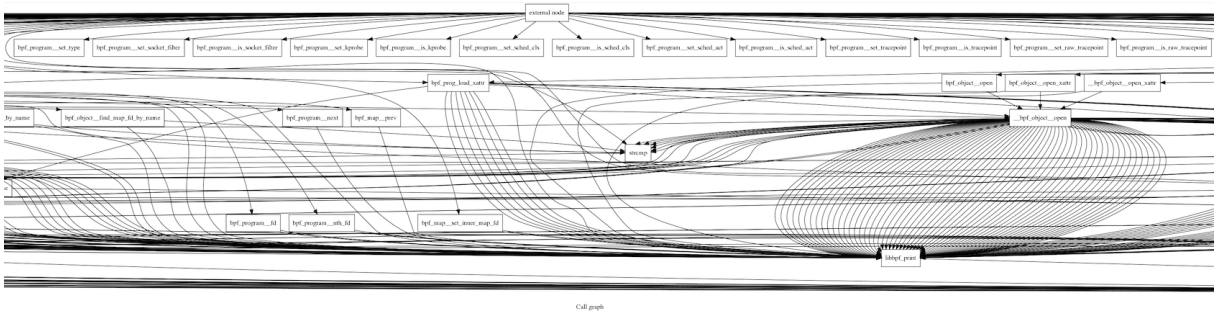

Figure 3.3. Part of the Call Graph for the Libbpf class.

Performing an exhaustive search requires an in-depth analysis on the call graph. Figure 3.3 was added to show the complexity of libbpf. It contains just a partial look at the libbpf call graph.

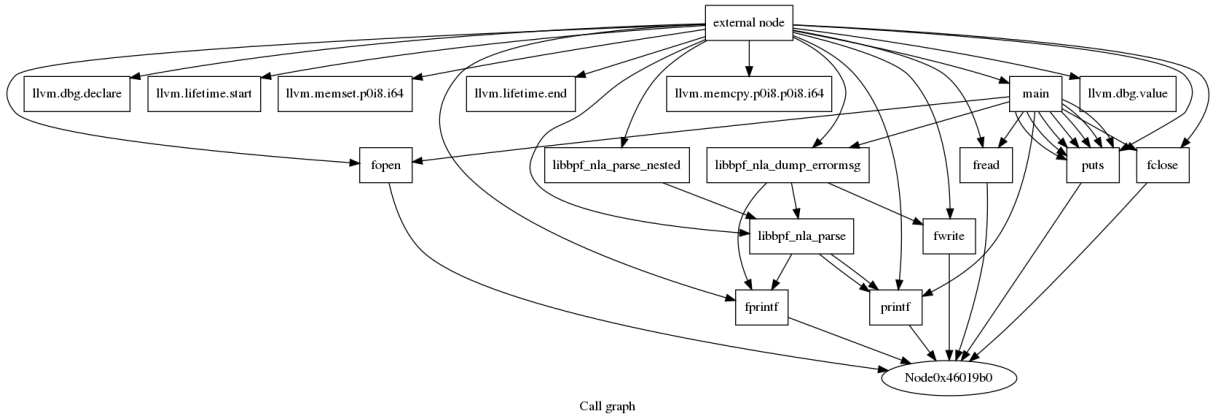

Figure 3.4. Call Graph for Nlattr Class

The call graph indicates both libbpf_nla_parse_nested and libbpf_nla_dump_errormsg

```
/**
* Create attribute index based on a stream of attributes.
* @arg tb     Index array to be filled (maxtype+1 elements).
* @arg maxtype  Maximum attribute type expected and accepted.
* @arg head   Head of attribute stream.
* @arg len    Length of attribute stream.
* @arg policy   Attribute validation policy.
*
* Iterates over the stream of attributes and stores a pointer to each
* attribute in the index array using the attribute type as index to
* the array. Attribute with a type greater than the maximum type
* specified will be silently ignored in order to maintain backwards
* compatibility. If \a policy is not NULL, the attribute will be
* validated using the specified policy.
*
* @see nla_validate
* @return 0 on success or a negative error code.
*/

printf("libbpf_nla_parse\n");
libbpf_nla_parse(&pta.tb, pta.maxtype, &pta.head, pta.len, &pta.policy);
```

Figure 3.5. NLA Parse Method Arguments

call libbpf_nla_parse, however the libbpf_nla_parse method can be called on its own. Also, the arguments required for this method are not modified in other methods, hence the order called does not matter. We can focus on letting the fuzzer focus its work on the 5 different inputs with its byte level-tracking.

Our driver program selects the top-level methods, those included in the standard libbpf library for the user, and determines a number of times to call them in a single program run. The BPF ecosystem relies on a few key data structures such as BPF maps and

```
bpf(BPF_MAP_CREATE, &bpf_attr, sizeof(bpf_attr));
```

Figure 3.6. BPF Syscall

```
bpf_map_lookup_elem(map_fd, void *key)
bpf_map_update_elem(map_fd, void *key, void *value)
bpf_map_delete_elem(map_fd, void *key)
```

Figure 3.7. BPF Map Methods

BPF programs, with all the helper functions in libbpf responsible for transforming said structures for the Linux Kernel.

Our driver program is designed to fill basic parts of BPF structs and let Angora choose the way in which they get called. Such parts of a BPF program include the BPF program type and attach type. The driver program combines this information and uses it for arguments in the bpf object load libbpf method. This returns a proper BPF object that can be used as an argument for other libbpf methods. This ensures all bugs found are valid bugs created. This approach proved to be much more successful compared to the previous program generative model.

3.2.1 BPF Map Generation

A large part of the BPF functionality revolves around the use of BPF maps. These maps are user defined data structures created in user space and utilized with BPF syscalls. These maps can then be accessed in user space or from the kernel-side via BPF helper functions. We will attempt to run these helper functions in user-space to efficiently fuzz without dealing with the drawbacks of working within the kernel.

BPF Commands can be constructed in userspace to contain a series of these BPF Map helper calls. See table 3.2 for a list of commands.

Figure 3.7 has some examples of user-space BPF Map helper functions. All variables passed to these methods are created by the user.

Table 3.1. BPF Program Types

BPF_PROG_TYPE
BPF_PROG_TYPE_UNSPEC
BPF_PROG_TYPE_SOCKET_FILTER
BPF_PROG_TYPE_KPROBE
BPF_PROG_TYPE_SCHED_CLS
BPF_PROG_TYPE_SCHED_ACT
BPF_PROG_TYPE_TRACEPOINT
BPF_PROG_TYPE_XDP
BPF_PROG_TYPE_PERF_EVENT
BPF_PROG_TYPE_CGROUP_SKB
BPF_PROG_TYPE_CGROUP_SOCK
BPF_PROG_TYPE_LWT_IN
BPF_PROG_TYPE_LWT_OUT
BPF_PROG_TYPE_LWT_XMIT
BPF_PROG_TYPE_SOCK_OPS
BPF_PROG_TYPE_SK_SKB
BPF_PROG_TYPE_CGROUP_DEVICE
BPF_PROG_TYPE_SK_MSG
BPF_PROG_TYPE_RAW_TRACEPOINT
BPF_PROG_TYPE_CGROUP_SOCK_ADDR

Table 3.2. BPF Commands

BPF_CMD
BPF_MAP_CREATE
BPF_MAP_LOOKUP_ELEM
BPF_MAP_UPDATE_ELEM
BPF_MAP_DELETE_ELEM
BPF_MAP_GET_NEXT_KEY
BPF_PROG_LOAD
BPF_OBJ_PIN
BPF_OBJ_GET
BPF_PROG_ATTACH
BPF_PROG_DETACH
BPF_PROG_TEST_RUN
BPF_PROG_GET_NEXT_ID
BPF_MAP_GET_NEXT_ID
BPF_PROG_GET_FD_BY_ID
BPF_MAP_GET_FD_BY_ID
BPF_OBJ_GET_INFO_BY_FD
BPF_PROG_QUERY
BPF_RAW_TRACEPOINT_OPEN

Our driver program for the BPF Maps enables us to create BPF Maps and dynamically adjust them with the fuzzer to further explore kernel side helper methods.

3.2.2 Valid Argument Generation

Another part of my fuzzing expedition within variable method calling includes valid argument generation. Many BPF methods take in an argument that was generated from a previous method. For example, many BPF methods take in a BPF_prog variable. There are some BPF methods that return a pointer to this BPF_prog variable. This is taken into account when building my programs containing several BPF methods. My engineered programs contain ways to test BPF methods with arguments created entirely from the fuzzer and arguments from existing BPF methods.

3.3 Syntactically Correct BPF Instruction Generation

Another challenge faced when fuzzing the BPF ecosystem is the generation of semantically correct arguments. In order to avoid a large portion of the fuzzer's time being rejected by the libbpf load program method, we designed a valid syntax generator on top of Angora to hopefully increase the speed of code coverage. Our generation engine is built on top of our driver class, and uses Angora's branch coverage logic and byte-level tracking to make decisions on future mutations. There is a fine balance between syntactically valid instructions and random instruction generation. Typical C programs are known to have the most crashes when inputs are entered that the user was not expecting. Therefore we propose a hybrid solution that can switch between inserting potential bug triggering - invalid syntax instructions and producing syntactically valid instructions to further explore the code paths.

Alongside syntactically correct instruction generation follows correct libbpf method argument construction. Methods in the driver program are designed to accept arguments from other methods responsible for creating the required structure type. This limits

Table 3.3. Table of example BPF branch instructions

opcode	Mnemonic	Pseudocode
0x05	ja + off	PC += off
0x15	jeq dst, imm, +off	PC += off if dst == imm
0x1d	jeq dst, src, +off	PC += off if dst == src
0x25	jgt dst, imm, +off	PC += off if dst > imm
0x2d	jgt dst, src, +off	PC += off if dst > src
0x35	jge dst, imm, +off	PC += off if dst >= imm
0x3d	jge dst, src, +off	PC += off if dst >= src
0xa5	jlt dst, imm, +off	PC += off if dst < imm

Table 3.4. Table of example BPF memory instructions

opcode	Mnemonic	Pseudocode
0x18	lddw dst, imm	dst = imm
0x61	ldxw dst, [src+off]	dst = *(uint32_t) (src + off)
0x69	ldxh dst, [src+off]	dst = *(uint16_t *) (src + off)
0x71	ldxb dst, [src + off]	dst = *(uint8_t *) (src + off)
0x79	ldxdw dst, [src+off]	dst = *(uint64_t *) (src + off)
0x62	stw [dst+off], imm	*(uint32_t *) (dst + off) = imm
0x6a	sth[dst + off], imm	*(uint16_t *) (dst+off) = imm
0x72	stb[dst+off], imm	*(uint8_t *) (dst + off) = imm

the amount of false-positives generated when the driver program randomly fills a BPF program that contains a variable otherwise not possible by existing libbpf Program creator helper methods.

3.4 Challenges

There were a few notable challenges to this project. The creation of a perfect driver program is not easy. There were several instances of false positives found by the fuzzer

due to a flaw with my code. These false positives were eventually all fixed, however they took a considerable amount of time to iron out. The other time consuming challenge came from compiling and building these BPF programs. Many hours were spent finding ways to compile my BPF source code with Angora and linking the correct libraries so that I could call BPF functions from my driver program. Although there is no definitive research challenge presented in building and compiling BPF programs with Angora, it will still be noted as the big time consuming part of the work done for this thesis.

Chapter 4

Results

4.1 Bug Report

After 48 hours of fuzzing each driver program, we discovered 8 BPF methods with bugs within the libbpf ecosystem. These bugs have been sent to the corresponding Linux BPF developers. These developers have not yet been able to confirm or deny any of inputs to be bug causing inputs. Due to the complex nature of BPF programs, they would need to compile and build my code on their machine which takes time. Also due to the recent COVID-19 outbreak, any bug report testing may have been delayed.

Each unique bug is given its own figure and description below. We include the unique crash number for some figures from the fuzzer as well, however these unique numbers can be misleading as often times a simple program change can fix many "unique" bugs. Examples of bugs found include a segfault on a BPF Type Format (BTF) operation, a stack-overflow on a libbpf error method, and an out-of-bounds array index on an bpf_unload method. These BTF operations include function information for included subroutines. However further analysis must be done on the developer side to confirm how effective these bugs can be at crashing a system or creating a vulnerability. To create these bugs, one must use my driver program and the input seed produced by Angora. My driver programs take in the binary file input (seed), and fill the pta struct accordingly.

4.2 Bug Analysis

Here we provide information on the types of bugs found.

4.2.1 Segmentation Faults

Segmentation faults appeared to be the most common bug found.

```
bpf_prog_linfo__lfind_addr_func(&bpl, pta.addr, pta.func_idx, pta.nr_skip);
```

Figure 4.1. BPF Function

There is a way to give the bpf_address method in figure 4.1 a function address that causes an out of bound error resulting in a seg fault. This error is similar to the one discussed in our bug case study at the end. Although it is hard to pinpoint the exact cause of this bug, the pta.func_idx value can be altered to cause this crash.

```
/**
* Create attribute index based on a stream of attributes.
* @arg tb     Index array to be filled (maxtype+1 elements).
* @arg maxtype  Maximum attribute type expected and accepted.
* @arg head   Head of attribute stream.
* @arg len    Length of attribute stream.
* @arg policy   Attribute validation policy.

*  libbpf_nla_dump_errormsg(struct nlmsghdr *nlh);
*/
libbpf_nla_dump_errormsg(&pta.nlh);
```

Figure 4.2. BPF Function

Figure 4.2 shows a top level method in libbpf netlink class. Struct contains 5 __u16 variables. The total path count for exploring this method was 24. Total there were 8 unique crashes here creating a mix of segfaults and Bus Errors. Here the segfault was created when the input value of nla_len is 13619 (large enough to cause an out of bounds error).

We found an error with the method in figure 4.3. There was a segmentation fault

```
//LIBBPF_API int libbpf_strerror(int err, char *buf, size_t size);
LIBBPF_API int libbpf_strerror(pta.err, &pta.buf, pta.size);
```

Figure 4.3. BPF Function

when the given char buffer (pta.buf) was not big enough. The driver program was able to mutate an integer err, a char buffer and a size_t value that resulted in a program crash from a READ memory access.

I found a few segmentation faults with the method in figure 4.4.

```
//bpf_object__find_map_by_name(const struct bpf_object *obj, const char *name);
bpf_object__find_map_by_name(&pta.obj, &pta.name);
```

Figure 4.4. BPF Function

A segmentation fault was found when a particular BPF object and character name was mutated by Angora. Calling this find by name method resulted in a crash likely due to a lack of error checking.

I found an error with the method in figure 4.5.

```
libbpf_nla_parse_nested(pta.nla_len, ...);
```

Figure 4.5. BPF Function

There was an input that caused the memset() function in the linux kernel to read an invalid negative size parameter. A value of nla_len in this instance is 64052. This value was not checked before the BPF parsing method called the memset function, leading to a stack overflow.

```
//libbpf_nla_dump_errormsg(struct nlmsghdr *nlh);
libbpf_nla_dump_errormsg(&pta.nlh);
```

Figure 4.6. BPF Function

4.2.2 Stack Smashing

I found a total of 23 crashes in the following method in figure 4.6. This bug was found by allowing Angora to fill the above nlmsghdr struct. There was a mix of segmentation faults and stack smashing. It should also be noted that there was a high path count (69) indicating that the fuzzer was thoroughly exploring the code.

4.2.3 Function Ordering

I was able to find a bug when calling the method in figure 4.7 twice in a single succession.

```
bpf_object__load(&bo);
.
.
bpf_object__load(&bo);
```

Figure 4.7. BPF Function

After digging through documentation to find if this was a problem for anyone else, I found a commit message describing how they currently needed to wait before calling a bpf_load() event twice, however given the unclear nature of this documentation and harsh segmentation fault instead of a graceful failsafe message, I count it as a bug for now. My method of variable method calling was able to catch this segmentation fault, even if the developers may be aware of this design flaw. Although I do not have the path count information available for this run, a similar bpf_object__open method was fuzzed with a path count of 90.

4.2.4 Bug Discussion

Fuzzing is one of the few ways to check the BPF for bugs. Other bug detection techniques such as concolic execution in combination with our driver program could also have been used to find these bugs. One new program in particular, MemLock [13], would have been a great candidate for this work as well. MemLock is specialized in choosing inputs that trigger a higher memory usage, and would have been great with our driver programs that create large buffers and BPF map creation. More information on MemLock can be seen in our related work section below.

4.3 Path Coverage Case Study

To further validate our claims of code coverage, we will analyze how well our BPF fuzzing expedition explored the source code. In figure 4.8, we have our BPF Object Check method which takes in a BPF object struct mutated by Angora. Our experiments were able to find all possible paths on this method.

It should also be noted that when fuzzing the method in figure 4.9, I encountered 65 paths explored on the driver program. A default of 54 possible paths exist in this particular driver template, leaving a path count of 11 for the create map node method. This appeared to be a complete exploration of this method.

4.4 Bug Case Study

Let's take a look at how some BPF fuzzing expeditions can be optimized. Certain methods don't require any changes from other methods, hence can be called in any particular order without worrying about correctness. The Netlink class within the libbpf library can be a great example of this.

Unloading a specific bpf_program in figure 4.10 results in a stack buffer overflow. There was an out of bounds access error because the fds array was not the length the program expected.

```
bpf_object__check_endianness(struct bpf_object *obj)
{
   static unsigned int const endian = 1;
   switch (obj->efile.ehdr.e_ident[El_DATA]) {
   case ELFDATA2LSB:
      if(*(unsigned char const *)&endian != 1)
         goto mismatch;
      break;

   case ELFDATA2MSB:
      if(*(unsigned char const *)&endian != 0)
         goto mismatch;
      break;
   default:
      return -LIBBPF_ERRNO__ENDIAN;
   }
   return 0;

mismatch:
   pr_warning("Error: endiannes mismatch.\n");
   return -LIBBPF_ERRNO__ENDIAN;
}
```

Figure 4.8. Path Coverage Example

```
LIBBPF_API int bpf_create_map_node();
```

Figure 4.9. BPF Create Map

```
void bpf_program__unload(struct bpf_program *prog)
{
   int i;
   if(!prog)
      return;
   if(prog->instances.nr > 0){
      for(i = 0; i < prog->instances.nr; i++)
         zclose(prog->instances.fds[i]);
   }
   else if(prog->instances.nr != -1){
      pr_warning("Internal error: instances.nr is %d\n", prog->instances.nr);
   }
}
```

Figure 4.10. BPF Program Unload

4.4.1 Failed Syscall Fuzzing

There were some failed attempts with vulnerability detection on the syscall side of BPF Map creation. Before focusing on available helper methods that run in kernel space, I decided to run a series of syscall tests using BPF Map commands. I constructed a driver program capable of fuzzing any syscall with a BPF command or structure in any particular order. These calls included the ones below in figure 4.11.

```
syscall(__NR_bpf, BPF_MAP_CREATE, &pta.attr, pta.one);
syscall(__NR_bpf, BPF_MAP_LOOKUP_ELEM, &pta.attr, pta.two);
syscall(__NR_bpf, BPF_MAP_UPDATE_ELEM, &pta.attr, pta.three);
syscall(__NR_bpf, BPF_MAP_DELETE_ELEM, &pta.attr, pta.four);
syscall(__NR_bpf, BPF_MAP_GET_NEXT_KEY, &pta.attr, pta.five);
```

Figure 4.11. BPF Syscall

```
struct bpf_insn {
   __u8 code; //opcode
   __u8 dst_reg:4; //dest register
   __u8 src_reg:4; //source register
   __s16 off; //signed offset
   __s32 imm; //signed immediate constant
};
```

Figure 4.12. BPF Instruction

These attributes passed in as arguments were mutated by Angora by a machine utilizing 80 cpu cores over a period of over 100+ hours. Unfortunately this brute force approach was unsuccessful at uncovering any new vulnerabilities. It appears Angora cannot properly instrument the code beneath the syscall barrier preventing it from fully utilizing the strengths of Angora. Therefore this approach was essentially a brute force blind attack using a large amount of CPU resources and time.

After the BPF_MAP Syscalls I further tried to brute force the BPF syscalls, this time I launched the BPF Syscalls with generated BPF programs via the BPF_PROG_LOAD operator. The bpf_insn variable that gets passed into the syscall has the following structure:

My first implementation had Angora mutate all these data values seen in figure 4.12, and load them as arguments in the BPF syscall. Unfortunately this method did not lead to any new bugs either. The same amount of CPU power and time was spent. Lastly, I created a system that enabled Angora to create only a series of valid instructions. Naturally, it is hard for Angora to mutate a series of bytes and have it form a semantically correct program that the syscall accepts. The previous brute force design had a problem of forming too many incorrect BPF instructions. These incorrect instructions were always rejected and thus not efficiently testing BPF. This new approach acts as a BPF Instruction generator that nearly guarantees the instructions are semantically correct. The fuzzer

determines which opcodes and registers it wants to use from a predetermined valid list, hence ensures most fuzzing loops are producing helpful inputs. These two combinations of random instruction generation and correct instruction generation were unable to produce good results on their own when loaded with the traditional BPF Syscall.

Chapter 5

Related Work

Earlier we discussed a fuzzing framework, bpf-fuzzer [1], that implements a mechanism to perform bpf program verification in user space. The project was started in 2015 and has found one bug since. More information on this project can be seen in chapter 2.

Setting up a fuzzer to fuzz code in kernel space can be a complicated ordeal. Unicorefuzz [8] proposes a novel way to fuzz parsers in the kernel space with a CPU emulator-based fuzzer. This is a coverage based fuzzer evaluated on synthetic and real world tests. Its main limitation is the execution speed 459 Execs/sec as compared to the baseline AFl at 4860 Execs/sec.

Syzkaller [10] is a coverage based fuzzer specialized to find bugs in the Linux Kernel. However it is now being extended to support other OS kernels. The syz-fuzzer engine sends inputs to the syz-executor to call syscalls and collects coverage via kcov. This is all performed in a virtual machine managed by syz-manager. They have found over 150 bugs in the Linux kernel alone.

Trinity [12] aims to reduce the number of invalid inputs for syscall fuzzing. This requires the user to supply annotation-type files to include information on arguments. This work has also found 150+ bugs inside the system call code. The work done for this thesis is similar due the creation of driver programs that form valid arguments for BPF calls.

KAFL [9] utilizes Intel's Processor Trace technology to fuzz with hardware assisted

code coverage. The Processor Trace data is then decoded and used to modify AFL's bitmap. This enables AFL to learn which inputs trigger new code coverage to mutate inputs accordingly.

Memlock [13] focuses on uncontrolled memory consumption bugs that lead to security weaknesses. The main idea is to select seeds from a seed pool that are more likely to keep increasing program memory. These memory consumption seeds are considered interesting and are given top priority to the mutation engine. They evaluated Memlock on 14 real world applications and had great results in finding memory related bugs.

IJON [4] was designed to give the user a more interactive experience with the fuzzer. They introduce the ability to add annotations to the source code to guide the fuzzer. These annotations take the form of functions from an included library that lets the fuzzer keep track of program states and disable coverage tracking. These annotations typically require just 1 - 4 extra lines of code to guide the fuzzer. The future work section below depicts more ways for the user to interact with a fuzzer.

Chapter 6

Future Work

This chapter will focus on providing a case study of recent fuzzing works and presenting future research ideas that coincide with our prior knowledge gained from our BPF fuzzing experience. There has recently been an increasing interest in the application of formal methods. We will take an in-depth look at how formal methods can improve and how we can apply the concepts learned earlier to improve the automatic verification of programs. First we will provide a look at related work.

6.1 Preventative Measures

In this section we will introduce techniques to further improve the parallel capabilities of fuzzing engines. In order to better understand the following techniques, we will look at a reason behind many software bugs.

6.1.1 Understanding Cause of Bugs

Studies have been done that indicate a majority of software bugs are caused by:

- Miscommunication - unclear requirements between testing teams
- Software Complexity - millions of lines of (object oriented) code
- Programming Errors - lack of unit tests
- Changing Requirements - unknown dependencies

- Time Pressures

Modern programs consist of millions of lines of code authored by numerous developers. Most code is never static, often receiving a plethora of updates over its lifetime. Modifying someone's code can have unforeseen consequences, leading to new bugs post update.

One of the main drawbacks to fuzzing is the hindrance of bugs on the exploration of the rest of the program. The complete process of fuzzing requires a thorough fuzz for 48 hours, fix all found bugs, and repeat. Many fuzzers such as AFL often focus on areas of code around a software bug. We performed a fuzzing test on a series of API calls with AFL, performing 2 fuzzing runs on an identical program. However program 2 had one additional API call at the end with a known bug. AFL had a higher path coverage on the first program without the additional buggy API call at the end, even though the second program had a higher potential path count to explore.

6.1.2 Potential Bugs

Regarding the problem of changing requirements, we have developed a method of fuzzing to combat this problem. Previously in the thesis, we discussed the finding of a bug B(5) in figure 1. We will refer to all bugs found in indirectly called methods (private methods not directly available to the user) as potential bugs. Potential bugs are not witness to real bugs. Many would dismiss this method as many false positives are produced, however we can often find merit in these false positives.

A change in program requirements often changes the way private methods are called. Developers utilize parts of the program in ways not thought of by the original author. A potential bug may be a false positive if a condition in a publicly available method keeps it from being triggered. However a change in requirements may have another developer trigger the "hidden" bug.

Therefore, we should theoretically be able to use potential bugs to guide the fuzzer to finding the real bug triggering input. However if the bug simply remains a potential bug, we can still utilize this information to help the developer account for future ways the method will be used. This leads us to our next section describing how we use these

potential bugs in a Formalistic way.

6.2 Fuzzing in a Formal Method Manner

Formal methods is a rigorous technique designed for the specification and verification of complex systems. A full formalization of a system is known to be a difficult task, requiring hours of rigorous mathematical proofs and a high level of knowledge of the software/ hardware design. The three general stages of Formal Methods are Specification, Development, and Verification. We propose Fuzzing as a lightweight hands-free approach to aid with the specification and partial verification of code.

Fuzzing already produces a witness to a program crash. However, it does not inherently tell the user other issues/concerns about the program. Formal methods often does more than verify the program is bug free - it guarantees correctness of program design behavior. As an example, it can verify whether a program always quits if $a > 9$. We propose that the power of fuzzing can be utilized in a new way to have more formalistic properties. These properties being specifications or requirements the developer needs to validate to a high degree of certainty.

First, the fuzzer should be given access to private/static methods not directly available to the user. These methods typically accept arguments that were crafted from previous methods, whether they be structs, arrays, pointers, or any other type of data. As mentioned earlier many program bugs often arise from changing requirements and unclear communication between the development team. One developer could be writing a private Method A with a certain use case in mind, then another Developer decides to call Method A with a different set of program parameters that causes the system to fail. Previously, this is only a problem that Formal Methods could potentially catch, but would go unnoticed by the fuzzer. Intel has previously utilized compartmentalized verification to ensure each individual piece of the chip is safe before assembly begins. Compartmentalizing software via individual method calling can potentially have the same benefits. Let's go over the base design.

A static / private method can receive a pointer to an object (*bpf_map). In order to

give the fuzzer direct access to this method, it will have to generate a correct bpf_map pointer. My previous works details the driver program responsible for creating a bpf_map object and assigning it a valid pointer. The method responsible for taking in this bpf_map pointer then receives it from the fuzzer/ driver program for its fuzzing executions. Note: the creation of a valid pointer to a real bpf_map struct removes the possibility of errors due to simple pointers with no valid address. If there is ever a possibility of the method receiving an invalid pointer due to regular use, then that is up to traditional fuzzing. The goal of this approach is to hit the method with all possible combinations of the data type to see where it fails. Let's say the target private method was designed to accept arguments that were checked to ensure the BPF Map had a valid title of size ($s < 100$). Typical fuzzing would never pass a value $s > 100$ into the static method because it is guarded by a condition. However future iterations of the code may not recognize this condition and pass a value of $s > 100$ to create a crash. Of course, this future update to the code can be fuzzed traditionally like the previous version, but often smaller code updates don't get the same fuzzing treatment as the first major implementation due to time constraints.

6.3 Rationalizing Invariant Work with Fuzzing

Fuzzing expeditions last for several days exercising billions of executions. These executions ideally explore all possible code paths to explore most state spaces of the program. Because of this, we have hours of CPU work before we find an input that crashes the system. I propose that we take advantage of all these program executions to allow the fuzzer to help the developer with program specifications and invariants. Discovering, creating, and enforcing program invariants has often been a difficult and time consuming task. While the fuzzer is busy at work exploring most possible state spaces of the program, it can generate a list of pseudo invariants for the developer. This allows the fuzzer to seamlessly generate code specification while proving it against millions of supporting program executions.

6.3.1 Important Security Invariant Generation with Angora

Our approach of dividing up a program into several slices can prove very helpful for finding potential "bug" invariants with the program. Going through source code and creating annotations can be a difficult and time consuming process requiring a high level of understanding about the inner workings of the program. Other recent work has gone into invariant generation using complex mathematical analysis and proofs.

Our method of finding "bug" invariants always warns the user of useful invariants. The invariants can be thought of conditions such as "if the variable x is ever equal to the value y, the program will crash". Not only are these invariants always useful, they are very helpful for future authors who will update the code. Having a generated comment section describing the variables that makes the method crash will help future developers avoid costly mistakes. This technique requires little to no knowledge of the source code's inner workings/ design. We also assume the people using this tool have access to the source code.

Traditional ways of fuzzing only start at the single point of entry BPF() and work their way down. This new program slice based technique creates a new program (separate from the original), that starts directly with the previously private method, allowing the fuzzer to feed directly into the private method.

Of course, the fuzzer on the programming slice has no knowledge of previous conditions on the input to this method. Instead we will use this knowledge to create our bug invariants. Once we find an input that triggers a bug, we add it to the collection of invariants. For example, if private method 1 crashed when the input was 9834574, then there would be a generated invariant over the private method 1 with a disclaimer about the bug indicating it has only been found by a program slice - not a full program execution.

Another use case for this bug invariant detection could be autonomous unit testing. While fuzzing inherently serves as an all in one complete unit test, these program slices can perform tests tailored to all specific methods. Perhaps the designer wanted to have a private method behave in a desirable way, and there was no time to create a unit test. Fuzzing alone may not find all the design flaws with the private method as they are hidden

```
for(int i = 0; i < 20; i++){
   if(i > pta.size){
      perror("Invariant Failed");
      raise(SIGSEV); //let the fuzzer track found invariants
   }
}
```

Figure 6.1. Invariant Check

behind earlier conditions that may change over time.

6.3.2 Merits to Fuzzing Based Invariants

Recent work has been done focusing on generation of loop invariants for program verification [7]. These invariants usually take many hours for programmers to construct and verify. Of course, if the developer wants a complete guarantee the code will work for a critical machine, formal methods and mathematical proofs are required. However, as demonstrated by the frequent use of fuzzers, many developers and programs are content with a very high percent chance of their code functioning if they can skip the intense formalistic process.

A fuzzer if combined with the right tools and techniques can end up being quite effective at generating free loop invariants.

I added the if-statement above to the for-loop in figure 5.1 to simulate a generated for-loop invariant. The raised error allowed me to track the fuzzer's time to calculate when the condition was satisfied.

AFL was able to find several cases of this invariant failing within the first 3 seconds of runtime on a single core. Therefore the invariant $i > pta.size$ was proven to be false.

```
File name is:
    ../afl-2.5b/loopInvariant1/crashes/id:000000,sig:11,src:00001,op:flip1
File was opened
Invariant failed
: Success
```

Figure 6.2. Fuzzer Output

We know know that "i" can be less than pta.size. In practice, you can place whatever invariant you want inside the loop with the raise() exception to allow the fuzzer to flag when the invariant turns false. If the fuzzer does not return any errors, the user can assume the invariant is true without having to perform any rigorous mathematical proof.

Another new invariant fuzzing could introduce is a stack/ memory based invariant. Typical invariants involve conditions on variables inside of loops such as $a > 5$. However recent fuzzing work from memlock [6] shows fuzzing being guided by memory consumption. Perhaps developers could learn to write invariants that include memory limits in recursive and for-loop constructs. The fuzzer could generate a warning over these dangerous functions giving the developers an early notice of memory issues.

State of the art fuzzing techniques have allowed us to dynamically explore programs like never before. With some modifications, fuzzers could turn into powerful tools for developers to gain further knowledge on their program. With millions of intelligent program executions, a fuzzer can say with a high degree of confidence that a program invariant stands correct or fails.

Chapter 7

Conclusion

With the Berkeley Packet Filter being a revolutionary new addition to the Linux kernel, it was imperative to analyze it for potentially fatal bugs. This thesis details the work required to perform a substantial fuzzing expedition over the BPF source code. Numerous driver programs were engineered to allow a fuzzer to create syntactically correct arguments for the BPF system calls and libbpf. The syntactically correct BPF instruction generation with variable method call fuzzing were effective in exploring BPF path code. Our results indicate the driver programs were successful in transforming binary input files to BPF arguments that thoroughly explored the BPF source code. The variable method call fuzzing was our most successful technique in regards to bug detection and path exploration. Our final analysis on state-of-the-art fuzzers creates a discussion on how to better improve a fuzzer's capability to assist with a developer's program analysis and verification workflow.

References

[1] BPF-Fuzzer. Available at `https://github.com/iovisor/bpf-fuzzer`.

[2] eBPF Offload Getting Started Guide. Available at `https://www.netronome.com/m/documents/UG_Getting_Started_with_eBPF_Offload.pdf` (2018).

[3] Cilium Authors. Revision 9b0ae85b. BPF and XDP Reference Guide.

[4] C. Aschermann, S. Schumilo, A. Abbasi, and T. Holz. IJON: Exploring Deep State Spaces via Fuzzing. In *IEEE Symposium on Security and Privacy (SP)*, pages 893–908, Oakland, CA, May 2020. IEEE Computer Society.

[5] Jesper Dangaard Brouer. eBPF - extended Berkeley Packet Filter. Available at `https://prototype-kernel.readthedocs.io/en/latest/bpf/`, 2016.

[6] P. Chen and H. Chen. Angora: Efficient Fuzzing by Principled Search. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 711–725, 2018.

[7] Geoff Hamilton. Generating Loop Invariants for Program Verification by Transformation. *Electronic Proceedings in Theoretical Computer Science*, 253:36–53, 08 2017.

[8] Dominik Maier, Benedikt Radtke, and Bastian Harren. Unicorefuzz: On the Viability of Emulation for Kernelspace Fuzzing. In *WOOT @ USENIX Security Symposium*, 2019.

[9] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels. In *USENIX Security Symposium*, 2017.

[10] syzkaller@googlegroups.com. Trinity. Available at `https://github.com/kernelslacker/trinity`.

[11] Liam Tung. Netflix: BPF is a new type of software we use to run Linux apps securely in the kernel. December 2019.

[12] Dimitri Vyokov. Syzkaller. Available at `https://github.com/google/syzkaller`.

[13] Cheng Wen, Haijun Wang, Yuekang Li, Shengchao Qin, Yichang Liu, Zhiwu Xu, Hongxu Chen, Xiaofei Xie, Geguang Pu, and Ting Liu. MemLock: Memory Usage Guided Fuzzing. In *International Conference on Software Engineering (ICSE)*, 2020.

[14] Michal Zalewski. American fuzzy lop. Available at `https://lcamtuf.coredump.cx/afl/` (2016).