Masaryk University
Faculty of Informatics

Linux Kernel System Call Fuzzing

Master's Thesis

Rao Arvind Mallari

Brno, Fall 2018

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Rao Arvind Mallari

Advisor: Prof. Petr Svenda

Acknowledgements

I thank Prof Petr Svenda for his timely review of the work and critical feedback. I also am indebted to my wife Khyati, for enabling and supporting me in this endeavor.

Abstract

Exisiting Linux kernel system call fuzzing techniques involve passing of completely random or semi-random(using templates) values as parameters to system calls to find bugs and corner cases. State of the art fuzzers successfully utilize evolutionary coverage guided techniques to improve the quality of the generated random values. Such techniques however do not detect deep bugs in the kernel. System calls, like any other API, is highly interdependent. Fuzzing such interdependent relations between the system calls require the creation of a valid context. This valid context is created when a sequence of system calls is made. The thesis builds upon the work of an existing coverage guided Linux System call fuzzer, TriforceAFL[1], by modifying it to support value passing in a sequence of system calls. A novel method utilizing deep learning and assisted regular expression parsing to generate high quality seeds for the modified fuzzer is also presented. Compared to the original TriforceAFL fuzzer, the modified fuzzer is able to cover more code paths and find more bugs in the kernel. The developed framework was also utilized to fuzz a proprietary DRDO Linux Kernel(DLK). The fuzzing framework was able to discover a previously undetected buffer overrun bug in the implementation.

Keywords

security, software testing, fuzzing, kernel fuzzing, coverage-guided fuzzing, system call fuzzing, . . .

Contents

1 Introduction

Vulnerabilities in operating systems are of utmost importance, since attackers can abuse these vulnerabilities to achieve full control of a system. System call fuzzing is a practical approach to discover bugs in an operating system kernel.

Current Linux kernel system call fuzzers fuzz a particular system call or family of system calls, using templates for parameters. Some state of the art fuzzers also incorporate coverage guided generation of random parameters. But the system call interface of Linux kernel, like any API, is stateful in nature. Hence the existing fuzzers have a high failure rate in terms of how many system calls are executed without an error[2]. Calling random system calls with random parameters wont work beyond a limit since there is no calling context. Take for example system calls of the networking subsystem in the Linux kernel. Calling `accept()` with random parameters will always fail. unless a call to a `socket()` is made to open a socket. Without considering such cases, it is unlikely that one will find bugs that are latent deep in the kernel[2].

The thesis proposes to improve an existing coverage guided Linux System call Fuzzer, TriforceAFL[1] by incorporating support for a value and order dependent system call sequence. This is achieved by utilizing deep learning techniques to build a model of ordering of system calls and regular grammar expressions with annotated system call definitions to infer a model of value dependence of system calls from a dataset of system call traces. The model is then used to generate candidate system call sequences which are utilized as seeds by the modified TriforceAFL fuzzer.

Chapter 2 of the thesis introduces the problem, lists some notable work related to this topic and provides a motivational example for pursuing this line of work. Chapter 3 introduces the technologies used to solve the problem statement. Chapter 4 describes the idea behind the proposed methodology and the challenges to realize them. The chapter also describes the realized design and explains the specifics of all the components. Chapter 5 of the thesis presents the details and results of the experiments performed. Chapter 6 concludes the thesis with closing remarks, possible improvements and future scope of the work done.

2 Problem Mapping

2.1 Linux Kernel

Linux is predominantly monolithic in nature, with many of the major subsystems like memory management, filesystems, networking and process management, all working in the same address space.

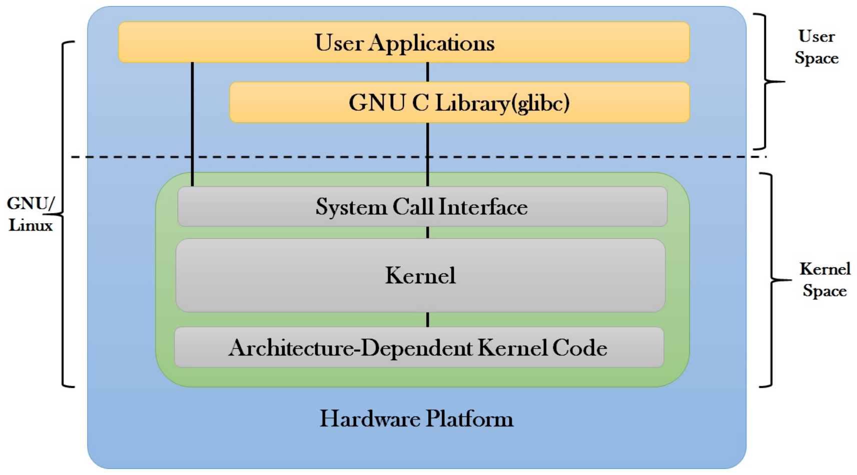


Figure 2.1: Linux Architecture

As shown in Fig. 2.1, the kernel divides the virtual address space into two parts[3], namely user space and kernel space. Applications like browsers, word processors, media players etc. run in the user space. While the operating system code and its subsystems, like process management, device drivers, network stack and memory management, resides in the kernel space. For purposes of code isolation and security, modern CPUs incorporate several privilege levels in which processes can reside. For example, Intel variant of processors offers four different levels of privileges. Linux utilizes two different modes, kernel mode and user mode, as shown in Fig. 2.2.

These two modes are builtin for the purposes of isolation and security. Kernel mode, or supervisor mode, is a higher privileged mode. Similarly the virtual address space is divided into two spaces, namely kernel space and user space. Fig. 2.2 shows this division in

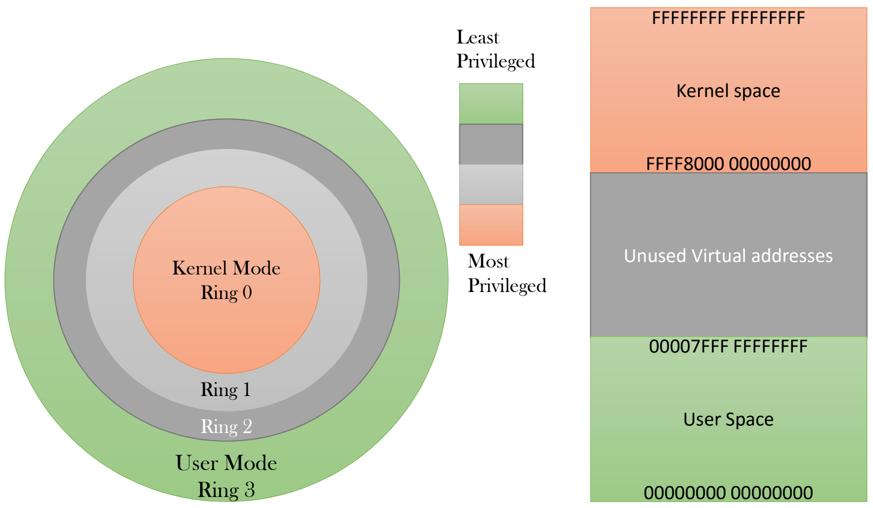


Figure 2.2: Isolation in the Linux kernel using privilege levels and virtual memory

a x86_64 Linux system. Code running in kernel mode resides in the kernel space and code running in user mode resides in user space. The user space is different for each process, while the kernel space is shared across all processes. In principle, a piece of code running in user mode is forbidden to access, execute or modify data/code in kernel space. Only code running in kernel mode from kernel space is allowed to perform such actions. Kernel mode code on the other hand can access and modify data residing in both kernel and user space. The idea behind such a design is to achieve isolation and non-interference.

2.1.1 System Call Interface

There are various situations wherein user mode process needs to perform a privileged operation, like manipulating I/O devices. But such a privileged operation can only be done in kernel mode. So the process running in user mode has to make a transition to kernel mode. Linux kernel allows this to do using a mechanism called system calls. The kernel exposes a thin interface, also known as system call interface, using which an user mode process can request the execution of a privileged operation. The kernel on receiving the request will perform checks for sanity, input validation and security. If such checks passes, the kernel will perform the privileged operation on the behalf of the requesting process. On completion of the operation, a transition is again made to user mode.

In an x86_64 system, an user mode process usessyscall instruction to make a system call. When making a system call, the arguments are passed using registers rdi, rsi, rdx, r10, r8, and r9 in that order. These parameters allow passing data between user mode and kernel mode code. Each system call has an integer identifying it. rax is set to the number of the desired system call and the syscall instruction makes the request to the OS kernel. The return value of the system call is stored in rax.

2.1.2 Kernel Fuzzing

The entire Linux Kernel is organized within a huge code base, with the lines of codes in the order of tens of millions. Auditing such a complex code using techniques like symbolic execution or source code auditing is a highly costly affair. The current best practice of fuzzing the kernel is by using system calls[4]. This is achieved by randomly calling system calls, also interchangeably referred to as kernel API, with randomly generated values as parameters with the aim of discovering borderline cases which wont be discovered by classical testing strategies.

2.2 Related work

According to the strategy employed, kernel fuzzers are categorised as : knowledge based fuzzers and coverage guided fuzzers[4].

Many of the kernel API functions are well documented with the number and type of the parameters being passed. This information along with some subjective information by the developer, is utilized by knowledge based fuzzers. Specifically, fuzzing a kernel using system calls need to cater to these two main problems: (1) the parameters are of specific data-types like structs, and hence the probability of creating well formed parameters by providing random bytes will be very less(2) Some kernel system calls need a context which is created by a prior call and hence the kernel API calls should be in a valid order[2]. Representative work include Trinity[5] and IMF[2]. Trinity is a type aware kernel fuzzer[5]. Trinity generates test-cases using the type information of parameters provided in the Linux system call prototype definitions. Data type of each parameter being passed to the system calls drives the generation of inputs from the initial corpus. To counter the problem of generation of invalid test-cases, Trinity also utilizes a list of constants being passed as parameters and range of values for certain variable parameters. IMF [2] observes the behaviour of a lot of executables for the system calls being executed and builds a dataset of a such sequences of system calls. These sequences are then utilized to to learn order dependence and the value dependence among API calls. The inferred model is then used in conjunction with a mutation engine to generate test cases. The authors manually annotated the parameters of API calls as in, out or inout to help in learning value dependence.

The recent trend in userland fuzzing is the use of gray box and coverage-guided fuzzing. The kernel developers have also tried applying these techniques to help in aiding to find kernel vulnerabilities. Representative work include syzkaller[6], TriforceAFL[1] and kAFL[7]. Syzkaller is a coverage-guided fuzzer for Linux kernels. It has found a high number of bugs in the kernel. Syzkaller uses the kernel with a patch and compiles it with a compiler option to measure the needed coverage data on execution. Syzkaller like Trinity[5] also incorporates targeted fuzzing by using type information of parameters of the system calls. Both coverage and bugs encountered are tracked in fuzzing sessions. Syzkaller, a project by Google, was the first ever open source kernel fuzzer which employed coverage-guided techniques in conjunction with templates of system calls. TriforceAFL is a modified version of AFL that supports kernel fuzzing using QEMU fullsystem emulation. This work converts the problem of kernel fuzzing to a file parser fuzzing by running the OS under a modified QEMU. AFL then fuzzes this process, monitoring for instruction pointer for any panics or crashes. By design, this fuzzer is able to fuzz any OS which can be emulated under QEMU[7]. KAFL utilizes a new feature available in the latest generations of the processors called Intel PT(processor trace). This hardware feature enables to measure coverage of code running from any defined location with minimal overhead. Since the code-coverage mechanism is built in the hardware, the speedup achieved when compared to measurement of code coverage of VMs in [1] was 40X. The authors in [8] demonstrate about how to fuzz Linux file system drivers by using an adapted version of AFL. This modified AFL version is based on compiling parts of the kernel using a modified compiler utilized in [6] to compute coverage during fuzzing sessions. This coverage information is shared with the fuzzer running in userland using a shared memory. The problem of this fuzzer is that it runs within realms of the targeted OS. Any bug, be it hang or crash, terminates the entire fuzzing session[9].

2.3 Motivational Example

The following is a code listing of an openly available proof of concept which triggers of the Linux kernel.

```
int main()
{
        int pipefd[2],result,in_file,out_file,zulHandler;
        loff_t viciousOffset = 0;
        char junk[30000]  ={0};
        result = pipe(pipefd);
        system("cat /dev/null > zul.txt");
        system("cat /dev/null > zug.txt");
        zulHandler = open("zul.txt", O_RDWR);
        memset(junk,'A',JUNK_SIZE);
        write(zulHandler, junk, JUNK_SIZE);
        close(zulHandler);
        viciousOffset = 0;
        in_file = open("zul.txt", O_RDONLY);
        result = splice(in_file, 0, pipefd[1], NULL,
                30000, SPLICE_F_MORE | SPLICE_F_MOVE);
        close(in_file);
        out_file = open("zug.txt", O_RDWR);
        viciousOffset =  118402345721856;
        // Create 108 tera byte file...
        //can go up as much as false
        //250 peta byte ext4 file size!!
        printf("ViciousOffset=");
        printf("%lu", (unsigned long)viciousOffset);
        //8446744073709551615
        result = splice(pipefd[0], NULL, out_file,
                &viciousOffset, JUNK_SIZE ,
                SPLICE_F_MORE | SPLICE_F_MOVE);
        if (result == -1){
                printf("Error-%s", strerror(errno));
                exit(1);
        }
        close(out_file);
        close(pipefd[0]);
        close(pipefd[1]);
        in_file = open("zug.txt", O_RDONLY);
        close(in_file);
        return 0;
}
```

The above code, when compiled and run, triggers a vulnerability(CVE-2014-7822 [10]) in the Linux kernel. The vulnerability exists in the splice() system call. splice() system call is of the form

```
ssize_t splice(int fd_in,
        loff_t *off_in,
        int fd_out,
        loff_t *off_out,
        size_t len,
        unsigned int flags);
```

splice() system call is used to move data between two file descriptors without copying between kernel address space and user address space. It transfers up to len bytes of data from the file descriptor fd_in to the file descriptor fd_out, where one of the file descriptors must refer to a pipe[11].

To trigger this vulnerability the attacker first creates a file zul.txt and fills it with 30000 A's. This file is then opened and spliced to one end of a pipefd. Then another file, zug.txt, is opened. This opened file used as a destination for a splice of the other end of the pipefd earlier opened. The second splice is made with size of 108TB! Such a huge size causes a memory corruption vulnerability in the Linux kernel and crashes the entire system. The vulnerability exists when an open() call is made for zug.txt.

2.4 Problem Statement

The above example highlights how a vulnerability is triggered only when a correct context is created. The main problem that this thesis aims to solve is the creation of this valid context. This context is created only when:

1. The system calls are made in the particular order.
2. Proper values of the parameters are passed to the system call.

The thesis aims to improve a kernel fuzzer, TriforceAFL[1], by attempting to create this context. To achieve this the thesis employs deep learning techniques and assisted regular expression parsing to create ordering and value dependencies.

3 Technologies Used

3.1 Fuzzing

Fuzzing is one of the most extensively used method to discover vulnerabilities in software. The basic idea is to continuously generate random inputs for the software under consideration, and observe the behaviour of the software on this input for buggy behaviour. By the use of fuzzing one can detect memory usage related bugs like use-after-free, buffer overflows, Uses of uninitialized memory, Memory leaks, Data races, deadlocks, NULL dereferences etc. and improper computation bugs like uncaught exceptions, div-by-zero, Memory exhaustion, hangs or infinite loops, infinite recursion (stack overflows) etc.

Fuzzing can be performed on a variety of target software. Following is a list(not exhaustive) of candidates that have been targeted by fuzzing over the years :

- Parsers of any kind (xml, json, asn.1, pdf, truetype, MS Office, jpeg, png etc.)
- Network protocols (HTTP, RPC, SMTP, MIME...)
- Crypto Libs(boringssl, openssl)
- Compilers and interpreters (Javascript, PHP, Perl, Python, Go, Clang,...)
- Browsers, text editors/processors (Chrome, vim, OpenOffice)
- OS Kernels (Linux), drivers, supervisors and VMs
- & much more like Media codecs, Compression utils, Datbases etc.

3.1.1 Types of Fuzzers

File-Format Fuzzers, as of today, can be broadly categorized into 3 broad categories based on the way random data is generated.

Blind Fuzzer

This type of fuzzer has absolutely no knowledge of the application being fuzzed. In this kind of fuzzing, the user generally collects a corpus of files, called input corpus to the fuzzer, which when given to the target application runs normally. The fuzzer generates new inputs to the application by changing the bytes of the files from the input corpus at random locations. This changing of byte values is called as mutations. Some blind fuzzers allow the users to control the amount of bytes that are being changed in 1 iteration. For example consider a simple elf parser. We provide the input corpus of 1 single file to the fuzzer whose byte representation is `7F 45 4C 46 02 01 01 00 00` .. ..

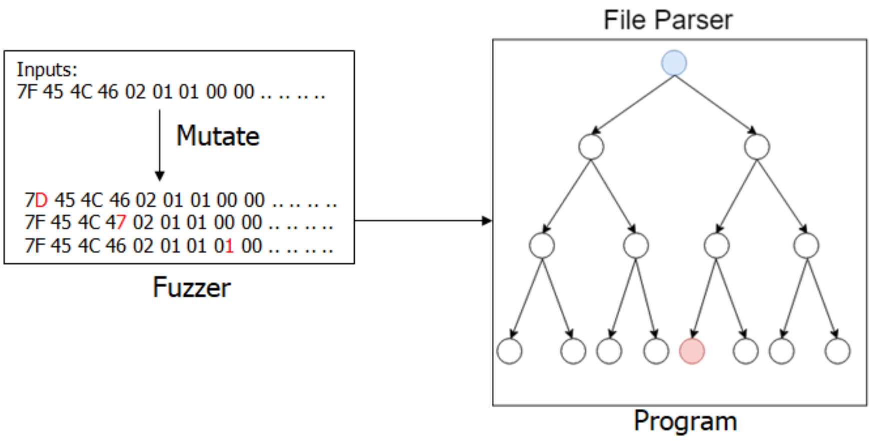

Inputs:
7F 45 4C 46 02 01 01 00 00 .. ..
Mutate
7D 45 4C 46 02 01 01 00 00 .. ..
7F 45 4C 47 02 01 01 00 00 .. ..
7F 45 4C 46 02 01 01 01 00 .. ..
Fuzzer
File Parser
Program

Figure 3.1: Blind Fuzzer

Fig. 3.1 depicts how a blind fuzzer works. As shown, the 1 file input corpus is mutated at random locations, one byte at a time(highlighted by red values) to generete new files. The newly generated files are then given as an input to the program. The tree like graph is a representation of the control flow graph(CFG) of the program, where each node is a Basic Block(BBL) of Execution. Blue nodes in the CFG represent the BBLs the program visits for a given input. Red node depicts the BBL in which there is a vulnerability, which we are interested to discover using fuzzing. The CFG shown is just a representation for explaining and comparing the working of the fuzzers. CFG for real programs have cycles and are much complex. As stated earlier, we are considering a simple parser. We can see that on execution of the mutated inputs the execution just visits the nodes in the edge. This is because of various file format structures like Transaction-Length-Value(TLV), Magic Number etc. are checked for the validity of input file in typical parsers. zzuf[12] and radamsa[13] are some famous fuzzers based on this principle.

Blind fuzzers are good for detecting shallow bugs and very easy to implement. The application doesn't need any modification for fuzzing. But the downside is that it takes a huge amount of iterations to detect bugs.

Grammar Based Fuzzer

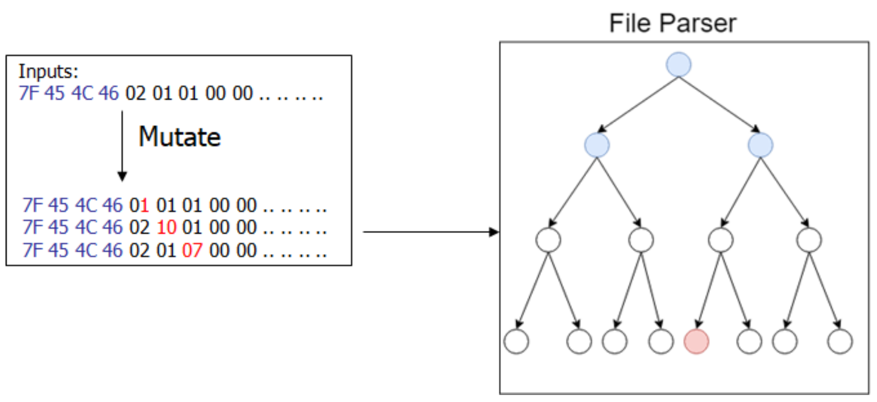

File Parser
Inputs:
7F 45 4C 46 02 01 01 00 00 .. ..
Mutate
7F 45 4C 46 01 01 01 00 00 .. ..
7F 45 4C 46 02 10 01 00 00 .. ..
7F 45 4C 46 02 01 07 00 00 .. ..

Figure 3.2: Grammar Based Fuzzer

The lacuna of blind fuzzers, was it wasted a huge amount of time because majority of mutated files were invalid as per the file format specification. To overcome this developers used grammars to generate the well-formed inputs, as well as to encode application-specific knowledge and test heuristics for guiding the generation of input variants[14]. This grammar guides the fuzzer to generate files which only conform to the grammar specifications. For our simple example, we consider the grammar that the first 4 bytes of the file are `7F 45 4C 46` (ELF Magic Header). Based on this grammar the files are mutated and fed to the file parser. As show in Fig. 3.2, we can see this type of fuzzers has better performance than the blind fuzzers. The generated files make the program visit nodes deeper than the one compared to blind fuzzers. This is because the grammar fuzzer will generate mostly well formed files which will pass all the initial checks done by the parser. Peach[15], packetdrill[16], csmith[17] and syzkaller[6] are some examples which uses grammar to generate inputs for the program.

Grammar specifications of file formats can be very complex. For example pdfv1.4[18] specification is approximately 1000 pages long. In some cases the grammar can also be proprietary[19]. Also correct files always doesnt mean better coverage. For our example of elf parser we can see that the generated files though will pass the initial checks of magic header, wont go deep down and find deep bugs occuring on rarely traversed code paths.

Evolutionary Fuzzers

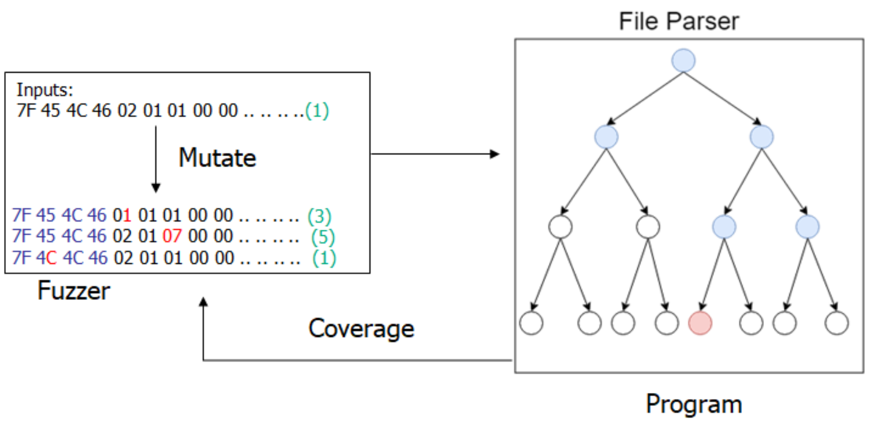

File Parser
Inputs:
7F 45 4C 46 02 01 01 00 00 .. ..(1)
Mutate
7F 45 4C 46 01 01 01 00 00 .. .. (3)
7F 45 4C 46 02 01 07 00 00 .. .. (5)
7F 4C 4C 46 02 01 01 00 00 .. .. (1)
Fuzzer
Coverage
Program

Figure 3.3: Evolutionary Fuzzers

Evolutionary Fuzzers are pretty much the state-of-art fuzzers as of today. Grammar Based-Fuzzers and Blind fuzzers did not observe how the program executed. They were only interested in the result of execution, namely no error, error or hang. Evolutionary fuzzers not only observe the result of execution but also observe the code path it took to reach the result. This information observed by evolutionary fuzzers is also called coverage. A coverage-based Evolutionary fuzzer aims at maximizing the code coverage to trigger paths that may contain bugs. To maximize code coverage, the fuzzer tries to generate inputs such that each input (ideally) executes a different path. Therefore, it is of paramount importance for a fuzzer to account for the gain obtained for each generated input[20].

In our example of elf parser, as shown in Fig. 3.3, we can see that the input was mutated at random locations(as shown by the red bytes). The green numbers in the brackets next to the inputs shows the coverage measured when the program was ran. We see that first two mutated inputs have more coverage and the third one has the same as the original input(because the first 4 magic bytes were mutated). The two new mutated inputs which have more coverage(coverage of 3 and 5) are added to the inputs and the next iteration proceeds. The inputs having no gain in coverage are discarded. This way the fuzzer will iteratively keep on mutating inputs which has more coverage to find more interesting inputs. Vuzzer[20], libfuzzer[21] and American Fuzzy Lop(AFL)[22] are some very famous fuzzers employing coverage driven evolutionary techniques.

Coverage-feedback driven evolutionary fuzzing has revolutionized the field of fuzzing. Also note how how the fuzzer learns the structure of file format due to the coverage feedback. In our case of elf, any mutation on the first 4 bytes doesnt yield coverage gain, since the magic is damaged and the parser will fail in the initial checks. So any changes in the first 4 bytes will be rejected and only changes in subsequent bytes will be retained and carried forward.

3.1.2 Why Evolutionary Fuzzers work?

Consider a small code fragment and its CFG shown in Fig. 3.4. Here the `input[4]` is a 4-byte buffer which is passed as an input to the program. The job of the fuzzer is to exactly generate input="abcd", to discover the bug hidden inside the CFG.

A Blind Fuzzer, as discussed in 3.1.1, will need to guess correctly "abcd" from a total of $2^8 \times 2^8 \times 2^8 \times 2^8 = 2^{32}$(4.2 billion) possibilities.

Now consider the performance of a Grammar Based Fuzzer, as discussed in 3.1.1. Suppose the grammar we decide is that each byte

```
if input[0] == ‘a’ {
        if input[1] == ‘b’{
            if input[2] == ‘c’ {
                if input[3] == ‘d’ {
                    ###BUGGY_CODE###
                }
            }
        }
}
```

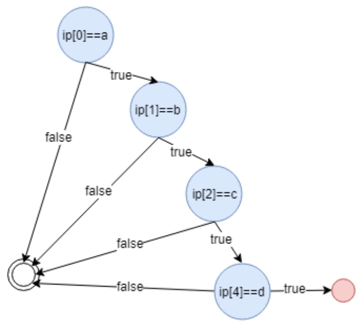


Figure 3.4: Code Fragment

can hold only printable ascii character. This means each byte can hold possibly 95 values. That means a grammar based fuzzer needs to guess "abcd" from a total of $95 \times 95 \times 95 \times 95 \approx 2^{26}$(67 million) possibilities. We have clearly reduced the search space as compared to the blind fuzzer, but the decision of the grammar to include only printable ascii is highly subjective and vaguely arbitrary.

Evolutionary Fuzzer on the other hand makes requires no such vague inputs. As disscussed in 3.1.1, Evolutionary fuzzer searches for inputs which maximizes coverage. The first coverage gain is found when a string of the form "a***" is guessed, where "*" can hold any one of the 2^8 values a byte can have. There are 2^8 strings of the form "a***". Once a string of form "a***" is found the fuzzer will discard all strings which mutates the first character, since changing the first character will lead to decrease in code coverage. Next, it has to guess "ab**" from 2^8 possibilities. Similarly for "abc*" and "abcd". So an evolutionary fuzzer will have to guess:

- "a***" from 2^8 possible values
- "ab**" from 2^8 possible values
- "abc*" from 2^8 possible values
- "abcd" from 2^8 possible values

In total the number of guesses are $2^8 + 2^8 + 2^8 + 2^8 = 2^{10} = 1024$. We can see the tremendously huge reduction of search space from a possible 4.1 billion values to a meager 1024 values by employing coverage guided evolutionary fuzzer with no vague or arbitrary grammar specifications.

3.2 AFL-Fuzzer

3.2.1 AFL-Fuzz Algorithm

AFL-Fuzzer [22] is one of the most recent and widely used coverage guided evolutionary fuzzer. AFL makes use of genetic algorithm and code instrumentation to achieve this. During fuzzing, AFL mutates the input file using a set of mutation techniques and generates a mutated input file. AFL executes the target program with this mutated input file and observes the code coverage. This mutated input file is considered interesting if the code-coverage observed is different than previously seen. This mutated input file is added to the set of files which are used to find new inputs with new code coverage in the next cycle(generation).

```
input: Seeds, Target Program P
Result: Malicious Inputs
while true do
    for $s \in Seeds$ do
        for $i \leftarrow 0$ to $k$ do
            candidate = Mutate(s);
            Coverage, Result = Run(Program, Candidate);
            if IsIncreased(coverage) then
                Add candidate to Seeds;
            end
            if result == crash OR hangs then
                Add candidate to results;
            end
        end
    end
end
```

Algorithm 1: AFL Fuzz Algorithm

Algorithm 1 shows the basic AFL-Fuzz algorithm which uses genetic algorithms and coverage to fuzz a target program. Here k depends on the size of s.

3.2.2 Measuring Coverage

AFL fuzzer uses binary rewriting technique to add instrumentation at the edge of each BBL to measure coverage. This is achieved by adding the assembly equivalent of the following pseudo code:

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

Here in the above code snippet, shared_mem[] is a 64KiB shared memory region which can be accessed from out side by another program(fuzzer). This shared memory buffer is used to share coverage data between the fuzzer and target program. Each location of the shared_mem array corresponds to the tuple (branch_src, brach_dst). The way its implemented we can see that the code paths $A \rightarrow B \rightarrow C \rightarrow D \rightarrow E$ is different than $A \rightarrow B \rightarrow D \rightarrow C \rightarrow E$, even though they visit the same BBLs in the CFG.

Binary rewriting is not the only method to calculate code coverage. There are other methods like Rewriting LLVM intermediate representation and using taint analysis with symbolic execution[23].

3.2.3 Mutation Strategies

AFL fuzz in its mutation function just doesn't do simple byte flips. There a collection of mutation techniques which are stacked upon each other stochastically. The mutations are either one or a combination of one of the following techniques:

- Sequential bit flips with varying lengths and stepovers
- Sequential addition and subtraction of small integers
- Sequential insertion of known interesting integers (0, 1, INT_MAX, etc)

- Stacked tweaks
  - Single-bit flips,
  - Attempts to set "interesting" bytes, words, or dwords (both endians),
  - Addition or subtraction of small integers to bytes, words, or dwords (both endians),
  - Completely random single-byte sets,
  - Block deletion,
  - Block duplication via overwrite or insertion,
  - Block memset.
- Test case splicing

For a detailed description of the mutation strategy the reader is advised to read more on the docs from the official repository of AFL[22].

3.3 TriforceAFL

3.3.1 Challenges in Kernel Fuzzing

Fuzzing kernels has a different set of challenges when compared to userland (or ring 3) fuzzing:

- Problem of detecting crashes and hangs of a kernel.
- Kernel code is much more complex than an user space application. The situation is made more difficult due to non-determinism due to interrupts, kernel level multi threading, statefulness, random response times from hardware etc.
- User space applications are generally fuzzed by providing random inputs. There is no such simple provision to provide inputs to the kernel.

TriforceAFL[1] overcomes all these problems by extending AFL's userland QEMU support to fuzz a VM running under QEMU's full-system emulation.

- The host OS runs QEMU and AFL. They communicate with each other through Unix pipes. The QEMU used is augmented with dispatch for a fake x64 instruction: *aflCall* (`0x0F24`). This fake instruction lets the guest VM perform a "hypercall" and communicate with the host.

- Guest VM running is the target. It also runs a small (userland) program called the agent, which makes the actual system call or sequence of system calls. The agent first pins a buffer to a fixed (physical) address. The agent then makes hypercall to tell AFL to put a serialized test case into this buffer. To AFL, this is just a buffer of bytes. On receiving the test case, agent uses a hypercall to tell QEMU to start recording an edge trace. Agent deserializes test case, and executes it. The test case then either causes a panic, or the agent makes a hypercall to indicate the test case executed normally.

In the most general case, TriforceAFL is a tool to allow AFL to find inputs that cause code executing in a VM to move to a basic block of interest[1].

3.3.2 Full Run Loop

1. TriforceAFL[1] boots a QEMU VM, waits till VM makes the aflCall startForkserver (so the agent needs to be invoked on guest startup)

2. The usual AFL loop of running a cycle of test cases, collecting feedback on the test case, and then creating a new generation of test cases at the end of the cycle still applies.

3. The forkserver creates a fork of the VM for each test case with a Copy-on-Write (CoW) view of memory, this lets each test case be isolated from each other! From here on out, everything happens in a fork of the VM.

    - The agent "wakes up" from its hypercall to startForkserver()
    - Makes a hypercall to getWork()
    - Deserializes the test case
    - Makes a hypercall to startWork() with the addresses to trace
    - Executes the test case
    - Either the test case crashes, or the agent makes a call to endWork() to signify the test case didn't crash

The fuzzer runs in a forked copy of the virtual machine, the entire in-memory state of the kernel for each test case is isolated. If the operating system uses any other resources besides memory, these resources will not be isolated between test cases. For this reason, its usually desirable to boot the operating system using an in-memory filesystem, such as a Linux ramdisk image[24].

3.3.3 Test Case File format

Test files use a proprietary format[25]. It supports multiple system calls made of of multiple arguments. Each file is made up of several sections separated by delimiters. The delimiters are designed to be mutable (so AFL can break delimiters) but not too likely to generate.

They are also intended to not often conflict with meaningful values, but this is a risk with the file format.

The file has a number of call records which are separated by `B7 E3` bytes. Each record describes a single system call and the call record is parsed in isolation.

A call record itself has a number of buffers separated by `A5 C9` bytes. The first buffer is special and contains the call header. The other buffers are referenced by the call header. A call record starts with a 16-bit number (all values are big-endian) which specifies the system call number. This is followed by exactly six arguments (whether or not the system call needs it).

Each argument starts with an 8-bit number specifying the type:

- Type 0 contains a 64-bit number that is used verbatim.
- Type 1 contains a 32-bit number that is used as an allocation size. The allocated buffer becomes the argument and the allocation size is pushed on a size stack.
- Type 2 has no further information. It consumes the next available buffer in the call record, and a pointer to the buffer becomes the argument. Its size is pushed on the size stack.
- Type 3 has no further information. It pops a size from the size stack and uses it as the argument.
- Type 4 consumes the next buffer and writes it to a temporary file. The file is opened and the file descriptor becomes the argument.
- Type 5 contains a 16-bit number that encodes a "file" type – it can represent sockets, special files, events, epolls, inotifys and other unusual file types. A file of that type is opened and the file descriptor becomes the argument.
- Type 7 starts with an 8-bit number specifying a count. An argument vector of this size is created by recursively parsing that many more arguments and storing them in the vector. The vector pointer becomes the argument.

- Type 8 has no further information. It consumes the next call buffer and writes it to a temporary file. The filename becomes the argument and the file size is pushed onto the size stack.
- Type 9 has an 8-bit number. When zero, the argument is the current process ID. When one the argument is the parent's process ID. When two, a new child process is forked (which does nothing), and the argument is the child's process ID.
- Type 10 contains two 8-bit numbers. The first references one of the earlier system call records, and the second references an argument number. The argument becomes a copy of the argument from the previous call.
- Type 11 starts with an 8-bit number specifying a count. A 32-bit argument vector of this size is created by recursively parsing that many more arguments and storing them in the vector. The vector pointer becomes the argument.
- Type 12 starts with an 8-bit number referencing one of the earlier system call records. The return value of that system call will become the argument.(not part of TriforceAFL, this is addition to the existing codebase)
- Type 13 has no further information. It selects a flag value at random from a list of all known flags.

3.4 Deep learning

Deep learning allows computational models to learn representations of data which have highly dependent temporal and spatial data. Deep learning has dramatically improved the state-of-the-art in speech recognition, visual object recognition, object detection and many other domains such as drug discovery and genomics[26]. When dealing with any sequences its of particular interest to look at Sequence to Sequence learning using Long Short-term Memory (LSTM)[27][28]. LSTMs are able to predict a long temporally correlated sequence, by predicting one step of the sequence at a time. LSTMs and other recurrent neural networks have been used extensively to generate sequence as complicated as musical notes[29][30], text[31] and images[32]. Due to the stochastic nature of neural networks, a well trained network can generate novel sequences with low error rates. Authors in [33] utilize Sequence to Sequence learning to generate API usage sequences for a given natural query. They use commented JAVA projects from the open source domain to build a dataset and learn a model. This model then predicts JAVA API usage sequence for a task defined by a query in natural language. Readily available implementations and access to faster hardware has made deep neural networks like LSTMs a viable and practical sequence generator.

4 Methodology

In this chapter, the central idea behind the solution of the thesis' problem statement is stated. Realized design of the solution based on the idea is also presented in the latter section of the chapter.

4.1 Idea

By utilizing AFL-Fuzzer in conjunction with a modified QEMU with custom hypercalls, TriforceAFL reduces the problem of coverage guided fuzzing of a kernel to that of the file parser. The target kernel, with the use of a ram file system and an agent, works like a file parser for files with format described in Section 3.3.3.

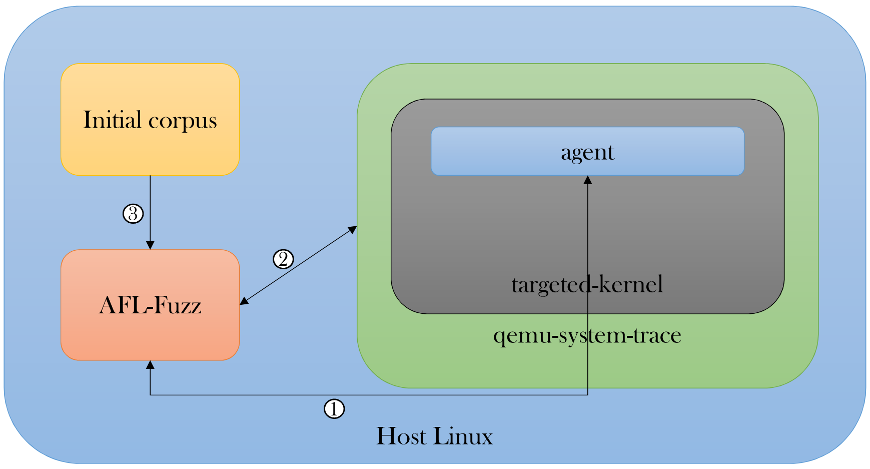


Figure 4.1: High Level Design

Fig. 4.2 shows a high level architectural design of how this idea is implemented by TriforceAFL. (1) denotes the start of VM, forking of VM using custom hypercall and submission of a fuzzed test case by AFL to the target kernel. (2) denotes the measuring of coverage when the kernel does the system calls with the parameters as described in the test case. AFL-Fuzz utilizes this coverage information and generates testcases$\textcircled{3}$ using genetic algorithm, as described earlier, to improve coverage and/or explore new paths.

In the current state, TriforceAFL is designed to test each system calls individually. Though it supports multiple system calls, it is not equipped to pass values across calls. Each system call has its own set of parameters with different datatypes. The thesis aims to enable TriforceAFL handle a sequence of system calls with value passing amongst the system calls in a sequence. To achieve this, the fuzzer has to be modified. The modified TriforceAFL is then provided with a set of files containing system call sequences as the initial corpus. These system call sequences are generated from a model, which is learnt and inferred from a dataset of observed sycall traces.

Following are the steps to realize the above mentioned idea:

1. Collect a set of applications.
2. Use strace and build a dataset of system call sequences.
3. Using the dataset built above and deep learning techniques, generate sequences of system calls conforming with both ordering and value dependence.
4. Use these sequences as initial testcases to modified TriforceAFL.
5. Run fuzzing sessions in modified TriforceAFL using these testcases.

4.2 Challenges

1. Curating a collection of applications for building a dataset which covers the entire breadth of system calls of the Linux kernel. There are some exotic system calls in the Linux kernel which are not called by many applications.
2. Learning the model of order and value dependence in the recorded traces and generating call sequences which will conform to this model.
3. Requisite development of TriforceAFL by introducing new code and structures to support value dependence.

4. Annotation of system call definitions to aid in model learning. The system call definitions' return values and parameters need to be annotated to enable more precise capturing of value dependence amongst the traces.

4.3 Realized Design

Generating a valid sequence of system calls automatically for use as seeds for fuzzing of Linux kernel is difficult due to the ordering and value dependencies amongst them. Producing system call sequences manually is a laborious task and requires huge amount of effort and knowledge. Instead the problem can be treated as an unsupervised machine learning task. By employing state-of-the-art deep learning techniques one can build models of system call sequences typically used by programs.

Fig. 4.2 shows a high level view of the components and its inter related nature in the realized design.

4.3.1 Recorded straces(system call traces)

The generative model needs to be first trained on an initial dataset which contains representational system call sequences. To create this initial dataset or seed corpus, strace was used. strace intercepts and records the system calls which are called by a process and the signals which are received by a process[34]. Fig 4.3 shows an example system call sequence captured using strace.

4.3.2 Extracting Ordering Dependencies

Encoder

The system call sequences as captured by strace needs to be encoded as numeric sequences for feeding input to the deep learning neural network. This encoding for a system call sequence is done using the system call number corresponding to each call in the sequence. For example a `mmap()` system call in the sequence is encoded as `9`. `arch/x86/entry/syscalls/syscall_64.tbl` file inside any modern

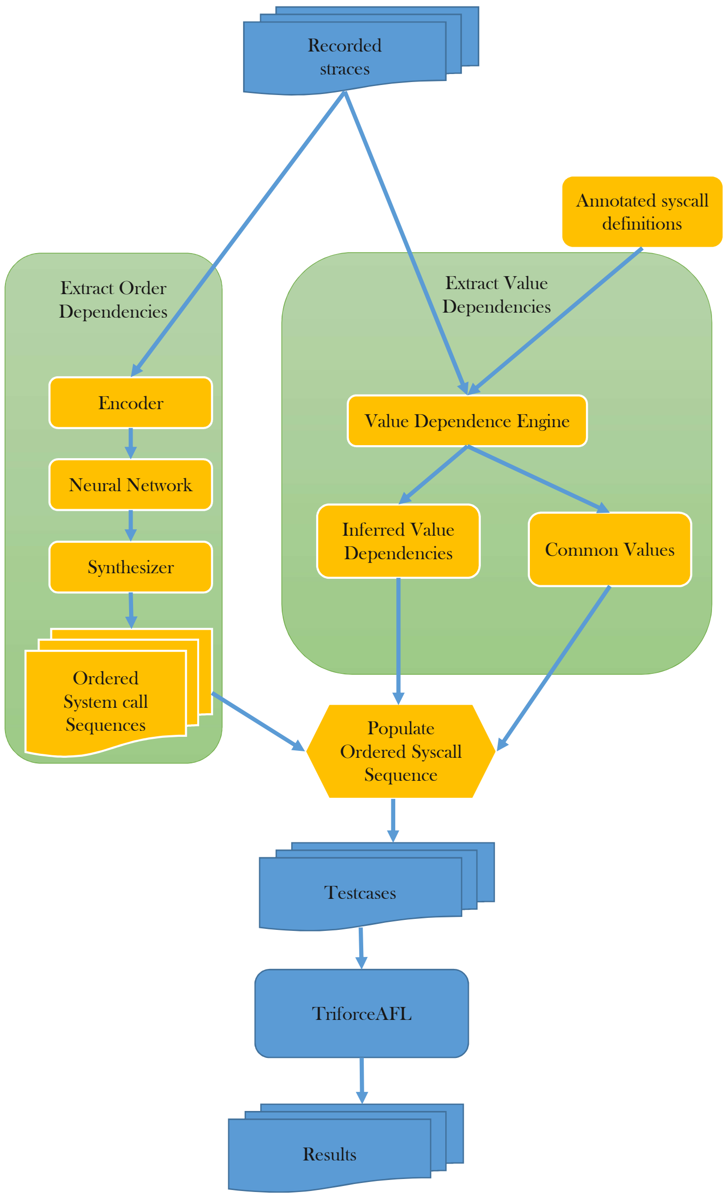

Recorded straces
Annotated syscall definitions
Extract Order Dependencies
Extract Value Dependencies
Encoder
Neural Network
Synthesizer
Ordered System call Sequences
Value Dependence Engine
Inferred Value Dependencies
Common Values
Populate Ordered Syscall Sequence
Testcases
TriforceAFL
Results

Figure 4.2: Realized Design

Linux kernel version source tree shows the number associated with each system call in that particular kernel version.

```
execve(0x7ffcac86d780, 0x7ffcac86eb40, 0x7ffcac86eb50) = 0
brk(0)                            = 0x191e000
mmap(0, 0x1000, 0x3, 0x22, 0xffffffffffffffff, 0) = 0x7f8f753e3000
access(0x7f8f751e1cd0, 0x4)       = -1 (errno 2)
open(0x7f8f751e05c7, 0x80000, 0x1) = 0x3
fstat(0x3, 0x7ffc3c6ddc30)        = 0
mmap(0, 0x1e82e, 0x1, 0x2, 0x3, 0) = 0x7f8f753c4000
close(0x3)                        = 0
open(0x7f8f753e3640, 0x80000, 0x7f8f753e6150) = 0x3
read(0x3, 0x7ffc3c6dddf0, 0x340)  = 0x340
fstat(0x3, 0x7ffc3c6ddc90)        = 0
mmap(0, 0x226950, 0x5, 0x802, 0x3, 0) = 0x7f8f74f9c000
mprotect(0x7f8f74fc0000, 0x1ff000, 0) = 0
mmap(0x7f8f751bf000, 0x2000, 0x3, 0x812, 0x3, 0x23000) = 0x7f8f751bf000
mmap(0x7f8f751c1000, 0x1950, 0x3, 0x32, 0xffffffffffffffff, 0) = 0x7f8f751c1000
close(0x3)                        = 0
open(0x7f8f753e3b08, 0x80000, 0x7f8f753e6150) = 0x3
read(0x3, 0x7ffc3c6dddc0, 0x340)  = 0x340
fstat(0x3, 0x7ffc3c6ddc60)        = 0
mmap(0, 0x204240, 0x5, 0x802, 0x3, 0) = 0x7f8f74d97000
mprotect(0x7f8f74d9b000, 0x1ff000, 0) = 0
mmap(0x7f8f74f9a000, 0x2000, 0x3, 0x812, 0x3, 0x3000) = 0x7f8f74f9a000
close(0x3)                        = 0
open(0x7f8f753e3fd0, 0x80000, 0x7f8f753e6150) = 0x3
read(0x3, 0x7ffc3c6ddd90, 0x340)  = 0x340
fstat(0x3, 0x7ffc3c6ddc30)        = 0
mmap(0, 0x1000, 0x3, 0x22, 0xffffffffffffffff, 0) = 0x7f8f753c3000
mmap(0, 0x208280, 0x5, 0x802, 0x3, 0) = 0x7f8f74b8e000
mprotect(0x7f8f74b95000, 0x200000, 0) = 0
mmap(0x7f8f74d95000, 0x2000, 0x3, 0x812, 0x3, 0x7000) = 0x7f8f74d95000
close(0x3)                        = 0
open(0x7f8f753c34b0, 0x80000, 0x7f8f753e6150) = 0x3
```

Figure 4.3: Sample strace output format used in the dataset

Neural Network

Long Short-Term Memory (LSTM) variant of Recurrent Neural Network is used infer ordering dependencies in system call sequences. By design, LSTM activations are dependent on both current value and previous values in a sequence. This property of LSTM enables inference of order dependence of system calls in a sequence. No information regarding the system call ordering dependency was provided to the neural network.

Ordered System Call Sequences

The trained network is then sampled to generate new system call sequences. The model is seeded with a random system call. The length of the sequence was selected randomly and can have a value from 1 to 16. Since longer system call sequences take longer time to execute, the sequence length chosen is small. Also a larger input file means we waste more fuzzing time searching for mutations in a larger search space.

4.3.3 Extracting Value Dependencies

Annotated System Calls

For enabling automated extraction of explicit value dependencies amongst the values returned and the parameters passed in a sequence of system calls, the return value and parameters of all system call definitions are annotated. The three annotations used are:

- __out__: A return value of a system call when annotated as __out__, implies that the return value of that system call can be used as a parameter in the following sequence of system calls.
- __in__: A parameter of a system call when annotated as __in__, implies that the return value of a previous system call in the sequence has to be used as the parameter.
- __flag__: A parameter of a system call when annotated as __flag__, implies that the value passed is of type flag. This annotation helps to generate more valid sequence of system calls.

As an illustration, the open and read system call is annotated as

```
__out__ int open(const char *pathname,
                 __flags__ int flags);

ssize_t write(__in__ int fd,
              const void *buf,
              size_t count);
```

Value Dependence Engine

Once the sequence of system calls is generated, the next task is to fill the parameters of these system calls keeping in mind that there can value dependencies amongst the calls. To learn the value dependencies the system calls definitions are annotated. These annotated system calls and recorded traces are passed to a Value Dependence Engine. Using regular expressions and annotated system call definitions, the value dependence engine extracts value dependence relations and flag values used. Whenever a system call in a trace is encountered whose return value is annotated as __out__, the value, system call number and its type is pushed on to the stack. Whenever in the trace, a system call with parameter annotated as __in__ is encountered, the value and type is compared with the elements of the stack. The current system call is then deemed to be dependent on the return value of the system call of the topmost element matching the value and type. Whenever in the trace, a system call with parameter annotated as __flag__ is encountered, the value is stored in a list.

Populate ordered System Calls

During generation of values of an ordered system call sequence, if a system call in the trace is encountered having its return value annotated as __out__ then the sequence number of system call in the trace and system call number is stored on a stack. Whenever a system call in the trace is encountered with parameter annotated as __in__ is encountered it checks the stack for any system calls its dependent on. The return value of topmost element on the stack which satisfies the value dependence relation determined earlier during model generation is passed as the value for the parameter. If no such element on the stack is found, the sequence is deemed to be invalid and discarded. If parameter of the system call in the trace is annotated as __flag__ then a flag value at random is selected from the list of flags and passed as the parameter.

Generate Testcases and Fuzzing

The system calls with parameter values were encoded using the file format of TriforceAFL. The encoded files are utilized as seeds for fuzzing using the modified TriforceAFL fuzzer. The results of the fuzzing excercises are a sequence of system calls with parameters which will trigger a bug or hang of the kernel.

5 Experiments and Evaluation

This chapter presents details the experiments and evaluation of the proposed fuzzing methodology.

5.1 Experimental Setup

5.1.1 Linux Kernels

Linux kernel version 3.5.5(x86_64), hereby known as vanilla Linux kernel, was used as the test kernel on which all fuzzing sessions were performed. The kernel was compiled with the default Kconfig and debugging symbols enabled. Proprietary Linux patches developed by DRDO were applied on the Linux kernel version 3.5.5, hereby known as DRDO Linux kernel(DLK). These patches increase the security of the kernel by implementing fine grained security controls inside the system calls. One of the major aims of this thesis was to check system calls of DLK for bugs.

5.1.2 Experimental Environment

The fuzzing sessions and deep learning was performed on a desktop workstation with Intel Core i5-6600k quad core CPU @ 3.50GHz, 16GB DDR4 RAM and NVIDIA GeForce GTX 1060 GPU. The operating system running on this workstation was RedHat Enterprise Linux 7.5. TriforceAFL and its Linux fuzzing logic of version 20160613 was taken from the original github repository. This version was then modified to support for passing of values across a sequence of system calls. The modified TriforceAFL is one of the important deliverables of this thesis.

5.1.3 Dataset Generation

strace version 4.12, provided in default installation of RHEL7, was used to collect system call traces. 1366 programs from the following sources were collected and run under strace for 10 times 1) Linux Testing Project (LTP) [35], 2) Linux Kernel selftests (kselftests) [36], 3) Standard Linux Binaries. The rationale behind recording the trace of each program multiple times is to capture some amount of the variability arising due to scheduling and multi threading.

The Linux Test Project is a joint project started by SGI, developed and maintained by IBM, Cisco, Fujitsu, SUSE, Red Hat and others, that has a goal to deliver test suites to the open source community that validate the reliability, robustness, and stability of Linux. The LTP testsuite contains a collection of tools for testing the Linux kernel and related features[35]. Out of the various available test programs, all of the 454 test programs from system calls section was traced using strace. The kernel contains a set of "self tests" under the tools/testing/selftests/ directory. These are intended to be small tests to exercise individual code paths in the kernel. Tests are intended to be run after building, installing and booting a kernel[36]. 49 programs from kselftests were chosen. A variety of programs are packaged with any Linux operating system in /bin directory. These programs behave differently depending on how the command line arguments are provided. Command line usage were extracted from example sections of respective man pages. 863 programs were selected for purposes of generating traces. These programs were then invoked with various command line arguments and switches as extracted from the man pages and documentaion.

5.1.4 Ordering Dependency Engine

The 13660 traces collected as part of Dataset Generation is then encoded as explained in Section 4.3.2. Encoded system call sequences are then fed as training input to the LSTM. The ordering dependency engine is written in Python3 and utilizes Keras for deep learning. A two layer LSTM network of 512 nodes per layer was used. The network is trained for 100 epochs. All other hyper parameters of the neural network are set to default values. The trained network is then sampled to generate random system call sequences.

5.1.5 Value Dependency Engine

The traces collected also are fed to the value dependency engine. This is also written in python and it utilizes regex matching and annotated system call definitions to extract value dependency and constant values.

5.1.6 Testcase Generation

The parameters of the ordered system call sequence generated by ordering dependency engine is then populated. The parameters values are selected at random so that value dependence is maintained. Also if the parameter value is flag value, a flag value at random is selected. The generated sequences with parameters value in TriforceAFL test case format is then used as a seed for purposes of fuzzing.

TriforceAFL also includes its own rudimentary system call generation. By manually analyzing the system call definitions from manpages, the authors have generated a set 116 system call shapes. For example (`fd`, `buffer`, `buffer`, `int`) is one of the shapes that a system call can have. For each system call the generator utilizes these 116 shapes to generate one testcase of each. The authors call this naive method of generating system calls with parameters as *kludging*. For our Linux version we have 341 Linux system calls. Hence 39556 system calls with parameters were generated.

Using the methodology of the thesis, 5000 system call sequences were generated to do a comparative analysis with TriforceAFL. 5000 inputs as seeds is too high a number. So 5000 system call sequences generated by the thesis' methodology and 39556 system calls generated by TriforceAFL's kludge method was minimized using minimization tool bundled with TriforceAFL called as *afl-cmin*. Each test case was ran and its coverage recorded. While minimizing a set of test cases, the tool will select minimal set of cases which will include the coverage of the entire set. 5000 system call sequences generated by the thesis' methodology were reduced to 413. While 39556 system calls generated by TriforceAFL's method was reduced to 287.

5.2 Evaluation

To evaluate modified TriforceAFL, which is one of the deliverables of the thesis, the minimized set of seeds were utilized to run 8 fuzzing instances for 18 hours each. 4 fuzzing instances ran the modified TriforceAFL with the 413 system call sequences as the seed. 4 instances ran the original TriforceAFL with the 287 system calls generated by kludging as the seed. The names of all files in the two minimized set of test cases were randomized in each run, since the alphabetical order of seeds affect the performance of fuzzing. The metrics discussed henceforth are the average of these 4 runs.

5.2.1 Comparison with TriforceAFL

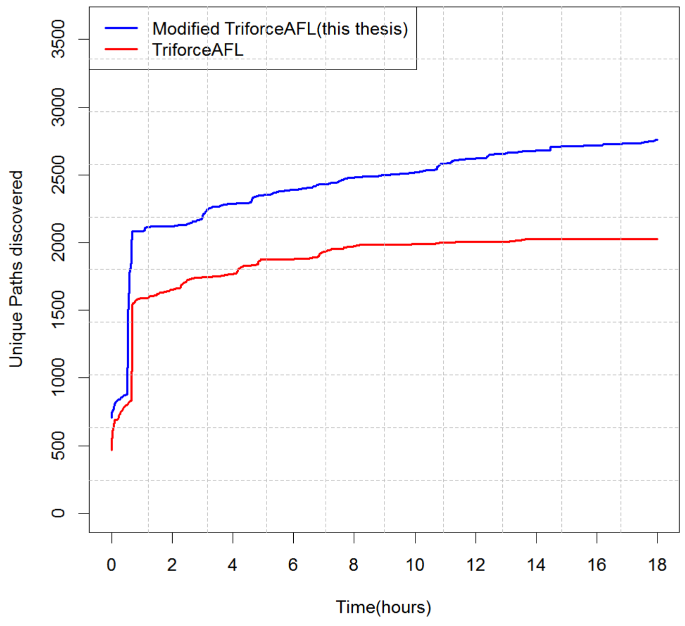


Figure 5.1: Comparison of unique paths discovered by both fuzzers

Figure 5.1 shows a comparison of the coverage performance between the unmodified TriforceAFL and the modified TriforceAFL. The coverage is measured in terms of paths in the CFG. It clearly shows a trend that the modified TriforceAFL is able to cover more paths. The modified TriforceAFL also has better initial coverage. This is because, the modified TriforceAFL utilizes a system call sequences as seeds to the fuzzing experiments. TriforceAFL utilizes only single system call sequence as seeds.

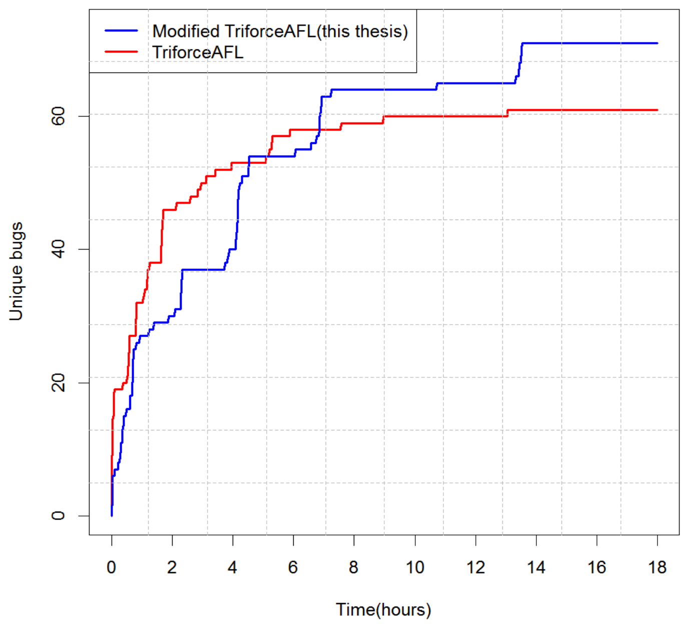
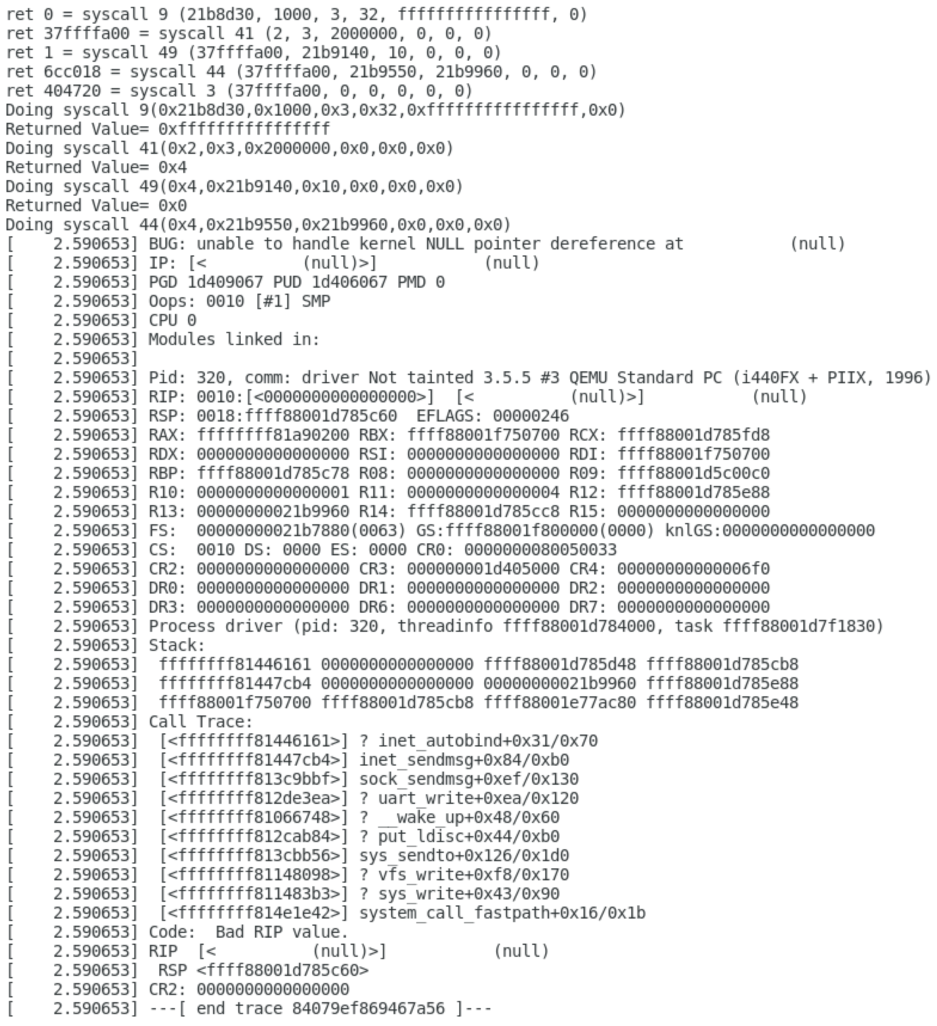


Figure 5.2: Comparison of unique bugs found by both fuzzers

A good fuzzer will have good coverage and bug discovery performance. Figure 5.2 shows a comparison of the bug discovery trends. The standard TriforceAFL algorithm provided good initial bug discovery. This can be attributed to the fact that standard TriforceAFL only has a single system call in all its seeds, while the modified TriforceAFL has a sequence as seed. Due to the fork server implementation, time taken to execute each test case is directly proportional to the number of system calls in the seeds. The standard TriforceAFL is able to mutate more and perform more executions and hence able to find more bugs initially. But this behaviour gets saturated pretty soon. Modified TriforceAFL keeps on discovering new bugs and eventually overtakes the number of bugs found by standard TriforceAFL. Out of 71 bugs found at end of 18 hours of fuzzing by modified TriforceAFL, 6 were kernel panics and remaining kernel hangs. Out of 61 bugs found at

18 hours of fuzzing by TriforceAFL none were kernel panics and all were kernel hangs.

5.2.2 Case Study of Bug Found



```
ret 0 = syscall 9 (21b8d30, 1000, 3, 32, ffffffffffffffff, 0)
ret 37ffffa00 = syscall 41 (2, 3, 2000000, 0, 0, 0)
ret 1 = syscall 49 (37ffffa00, 21b9140, 10, 0, 0, 0)
ret 6cc018 = syscall 44 (37ffffa00, 21b9550, 21b9960, 0, 0, 0)
ret 404720 = syscall 3 (37ffffa00, 0, 0, 0, 0, 0)
Doing syscall 9(0x21b8d30,0x1000,0x3,0x32,0xffffffffffffffff,0x0)
Returned Value= 0xffffffffffffffff
Doing syscall 41(0x2,0x3,0x2000000,0x0,0x0,0x0)
Returned Value= 0x4
Doing syscall 49(0x4,0x21b9140,0x10,0x0,0x0,0x0)
Returned Value= 0x0
Doing syscall 44(0x4,0x21b9550,0x21b9960,0x0,0x0,0x0)
[    2.590653] BUG: unable to handle kernel NULL pointer dereference at           (null)
[    2.590653] IP: [<          (null)>]           (null)
[    2.590653] PGD 1d409067 PUD 1d406067 PMD 0
[    2.590653] Oops: 0010 [#1] SMP
[    2.590653] CPU 0
[    2.590653] Modules linked in:
[    2.590653]
[    2.590653] Pid: 320, comm: driver Not tainted 3.5.5 #3 QEMU Standard PC (i440FX + PIIX, 1996)
[    2.590653] RIP: 0010:[<0000000000000000>]  [<          (null)>]           (null)
[    2.590653] RSP: 0018:ffff88001d785c60  EFLAGS: 00000246
[    2.590653] RAX: ffffffff81a90200 RBX: ffff88001f750700 RCX: ffff88001d785fd8
[    2.590653] RDX: 0000000000000000 RSI: 0000000000000000 RDI: ffff88001f750700
[    2.590653] RBP: ffff88001d785c78 R08: 0000000000000000 R09: ffff88001d5c00c0
[    2.590653] R10: 0000000000000001 R11: 0000000000000004 R12: ffff88001d785e88
[    2.590653] R13: 00000000021b9960 R14: ffff88001d785cc8 R15: 0000000000000000
[    2.590653] FS:  00000000021b7880(0063) GS:ffff88001f800000(0000) knlGS:0000000000000000
[    2.590653] CS:  0010 DS: 0000 ES: 0000 CR0: 0000000080050033
[    2.590653] CR2: 0000000000000000 CR3: 000000001d405000 CR4: 00000000000006f0
[    2.590653] DR0: 0000000000000000 DR1: 0000000000000000 DR2: 0000000000000000
[    2.590653] DR3: 0000000000000000 DR6: 0000000000000000 DR7: 0000000000000000
[    2.590653] Process driver (pid: 320, threadinfo ffff88001d784000, task ffff88001d7f1830)
[    2.590653] Stack:
[    2.590653]  ffffffff81446161 0000000000000000 ffff88001d785d48 ffff88001d785cb8
[    2.590653]  ffffffff81447cb4 0000000000000000 00000000021b9960 ffff88001d785e88
[    2.590653]  ffff88001f750700 ffff88001d785cb8 ffff88001e77ac80 ffff88001d785e48
[    2.590653] Call Trace:
[    2.590653]  [<ffffffff81446161>] ? inet_autobind+0x31/0x70
[    2.590653]  [<ffffffff81447cb4>] inet_sendmsg+0x84/0xb0
[    2.590653]  [<ffffffff813c9bbf>] sock_sendmsg+0xef/0x130
[    2.590653]  [<ffffffff812de3ea>] ? uart_write+0xea/0x120
[    2.590653]  [<ffffffff81066748>] ? __wake_up+0x48/0x60
[    2.590653]  [<ffffffff812cab84>] ? put_ldisc+0x44/0xb0
[    2.590653]  [<ffffffff813cbb56>] sys_sendto+0x126/0x1d0
[    2.590653]  [<ffffffff81148098>] ? vfs_write+0xf8/0x170
[    2.590653]  [<ffffffff811483b3>] ? sys_write+0x43/0x90
[    2.590653]  [<ffffffff814e1e42>] system_call_fastpath+0x16/0x1b
[    2.590653] Code:  Bad RIP value.
[    2.590653] RIP  [<          (null)>]           (null)
[    2.590653]  RSP <ffff88001d785c60>
[    2.590653] CR2: 0000000000000000
[    2.590653] ---[ end trace 84079ef869467a56 ]---
```

Figure 5.3: NULL Dereference Bug found by Modified TriforceAFL(this thesis)

Fig 5.3 shows how a NULL dereference bug in the kernel looks like. This bug is triggered by a sequence of system calls mmap(9), socket(41), bind(49), sendto(44) and close(3) in that order. The kernel crashes on a NULL pointer dereference when making the system call sendto(). As evident in the crash stack trace we can see that the the socket() system call with some flags returns a socket file descriptor(0x4). This socket file descriptor is then passed on to bind() system call. The bug is triggered when a sendto() system call is made on this socket(0x4). This bug demonstrates the efficacy of the methodology of the thesis. The crash is triggered only when the system calls are made in this sequence. Value passing amongst the dependent system calls is also required to reach this bug in the kernel. This crash was not discovered by any fuzzing session involving standard TriforceAFL.

5.2.3 Comparison of Vanilla Kernel with DRDO Linux Kernel

DRDO Linux Kernel(DLK) is a modified propreitary Linux kernel with the system call implementations modified to achieve mandatory access control. One of the major aims of the thesis was to check this added code for bugs and vulnerabilities. As part of development of DLK, test prgrams are written which evaluated the functioning of the added features of DLK. These test cases are specifically meant for the testing of features of DLK. A subset of these programs which targeted the exec() and execve() modifications were utilized for trace dataset generation. Similar to the fuzzing of vanilla kernel, a LSTM was trained using these traces. The value dependence relations and FLAG values used were extracted from these traces. 250 sequences of system calls were generated using this information, which were used as a seed for the fuzzing sessions.

The fuzzer was tweaked to only allow the system calls made by these test programs. This was done to focus the fuzzing sessions only to target paths which are modified as part of DLK. 8 instances of fuzzer were ran using the generated seed system call sequences. 4 instances were run against the DLK and 4 instances were run against the vanilla kernel. Following graph shows the comparison of the average unique bugs found.

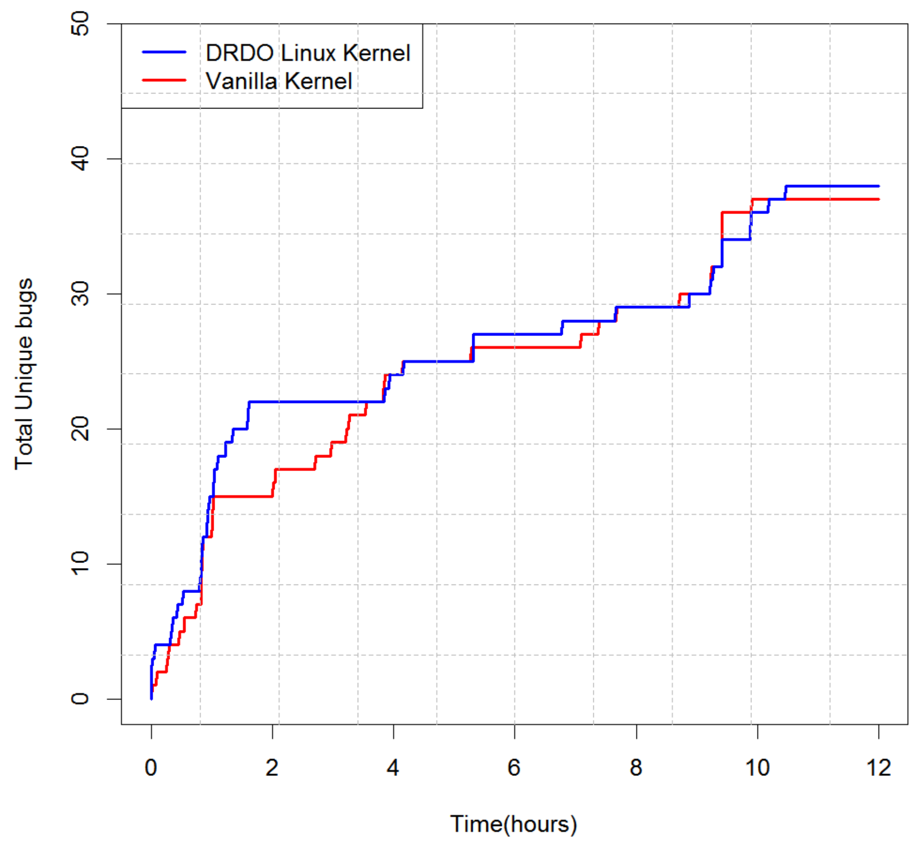


Figure 5.4: Comparison of Bugs found in DRDO Linux Kernel and Vanilla Kernel by Modified TriforceAFL(this thesis)

It can be seen that the bug discovery trend of both kernels are more or less same. Except one, all the bugs found affected both the kernels. The one bug which only affected DLK was a buffer overrun. The difference in the observed bug discovery trend is less since the modification introduced by DLK was also very less. Continuous fuzzing of DLK will increase the assurance level and improve code quality. Kernel Fuzzing using modified TriforceAFL has now been included as part of the standard testing strategy of DLK.

6 Conclusion and Future Work

The thesis designed, implemented and evaluated Linux Kernel System Call fuzzing using the methodology described in Section 4.3. The thesis shows how the performance of an existing coverage guided kernel fuzzer can be improved by employing deep learning to model ordering of system call sequences and assisted regular expression parsing of recorded system call traces to infer value dependence. The modified fuzzer is able to explore more paths and find better quality bugs. The case study of a bug found asserts the central idea behind the thesis of finding deeper bugs if a value and order dependent system call sequence is used as seed. The modified fuzzer was also used to test proprietary DRDO Linux Kernel(DLK). A previously unknown bug in DLK was also discoverd as a result. Just like fuzzing has been incorporated fully into software development life cycle of applications, the thesis concludes that kernel development can also leverage coverage gudide fuzzing techniques to improve code quality and assurance.

Future Work

Owing to the broad spectrum of the research area of the thesis, there is a huge scope of addition of new ideas and techniques. Some important directions of future work are:

- Incorporating compile time instrumentation to improve fuzzing introduced by enabling KCOV flag in Kconfig of any kernel.
- The current fuzzing framework is single threaded. This limits the types of bugs that can be found. A multithreaded agent with some measure of stochastic variability can yield bugs like race conditions and use after free bugs.
- The annotations of system call definitions only include explicit dependencies. Implicit dependencies, like pointers to buffers, are not catered to in the thesis. Any work in this direction may help find higher quality deeper bugs.

Bibliography

1. HERTZ, Jesse; NEWSHAM, Tim. *TriforceAFL*. 2017 (accessed 19 November, 2018). Available also from: `https://github.com/nccgroup/TriforceAFL`.
2. HAN, HyungSeok; CHA, Sang Kil. IMF: Inferred Model-based Fuzzer. In: *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. 2017, pp. 2345–2358.
3. BOVET, Daniel P; CESATI, Marco. *Understanding the Linux Kernel: from I/O ports to process management*. " O'Reilly Media, Inc.", 2005.
4. LI, Jun; ZHAO, Bodong; ZHANG, Chao. Fuzzing: a survey. *Cybersecurity*. 2018, vol. 1, no. 1, pp. 6.
5. JONES, Dave. Trinity: A linux kernel fuzz tester. In: *Linux Conf Australia*. 2013.
6. VYUKOV, Dmitry. *syzkaller - kernel fuzzer*. 2015 (accessed May 5, 2018). Available also from: `https://github.com/google/syzkaller`.
7. SCHUMILO, Sergej; ASCHERMANN, Cornelius; GAWLIK, Robert; SCHINZEL, Sebastian; HOLZ, Thorsten. kafl: Hardware-assisted feedback fuzzing for OS kernels. In: *https://www.usenix.org/system/files/conference/usenixsecurity17/sec17-schumilo.pdf*. 2017.
8. NOSSUM, Vegard; CASASNOVAS, Quentin. *Filesystem Fuzzing with American Fuzzy Lop*. Vault, 2016.
9. CHEN, Chen; CUI, Baojiang; MA, Jinxin; WU, Runpu; GUO, Jianchao; LIU, Wenqian. A systematic review of fuzzing techniques. *Computers & Security*. 2018.
10. *CVE-2014-7822*. [Available from MITRE, CVE-ID CVE-2014-7822.]. 2018(accessed 19 November, 2018). Available also from: `http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2014-7822`.
11. PAGES, man. *SPLICE(2) Linux Programmer's Manual*. 2017 (accessed 19 November, 2018). Available also from: `http://man7.org/linux/man-pages/man2/splice.2.html`.

12. HOCEVAR, S. *zzuf—multi-purpose fuzzer*. 2011 (accessed 19 November, 2018). Available also from: `https://github.com/samhocevar/zzuf`.

13. HELIN, Aki. *Radamsa fuzzer*. 2011 (accessed 19 November, 2018). Available also from: `https://gitlab.com/akihe/radamsa`.

14. GODEFROID, Patrice; KIEZUN, Adam; LEVIN, Michael Y. Grammar-based Whitebox Fuzzing. *SIGPLAN Not.* 2008, vol. 43, no. 6, pp. 206–215. ISSN 0362-1340. Available from DOI: `10.1145/1379022.1375607`.

15. EDDINGTON, Michael. Peach fuzzing platform. *Peach Fuzzer*. 2011, pp. 34.

16. CARDWELL, Neal; CHENG, Yuchung; BRAKMO, Lawrence; MATHIS, Matt; RAGHAVAN, Barath; DUKKIPATI, Nandita; CHU, Hsiao-keng Jerry; TERZIS, Andreas; HERBERT, Tom. packetdrill: Scriptable Network Stack Testing, from Sockets to Packets. In: *USENIX Annual Technical Conference*. 2013, pp. 213–218.

17. YANG, Xuejun; CHEN, Yang; EIDE, Eric; REGEHR, John. Finding and understanding bugs in C compilers. In: *ACM SIGPLAN Notices*. 2011, vol. 46, pp. 283–294. No. 6.

18. INCORPORATED, Adobe Systems. *PDF Reference, Third Edition, version 1.4*. 2001 (accessed 19 November, 2018). Available also from: `https://www.adobe.com/content/dam/acom/en/devnet/pdf/pdfs/pdf_reference_archives/PDFReference.pdf`.

19. KNOLL, Thomas. *Adobe Photoshop File Formats Specification*. 2016 (accessed 19 November, 2018). Available also from: `https://www.adobe.com/devnet-apps/photoshop/fileformatashtml/`.

20. RAWAT, Sanjay; JAIN, Vivek; KUMAR, Ashish; COJOCAR, Lucian; GIUFFRIDA, Cristiano; BOS, Herbert. Vuzzer: Application-aware evolutionary fuzzing. In: *Proceedings of the Network and Distributed System Security Symposium (NDSS)*. 2017.

21. SEREBRYANY, Kosta. Continuous fuzzing with libfuzzer and addresssanitizer. In: *Cybersecurity Development (SecDev), IEEE*. 2016, pp. 157–157.

22. ZALEWSKI, Michal. *american fuzzy lop*. 2014 (accessed 19 November, 2018). Available also from: `http://lcamtuf.coredump.cx/afl`.

23. 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. In: *NDSS*. 2016, vol. 16, pp. 1–16.

24. HERTZ, Jesse; NEWSHAM, Tim. *Project Triforce: Run AFL on Everything!* 2016 (accessed 19 November, 2018). Available also from: `https://www.nccgroup.trust/uk/about-us/newsroom-and-events/blogs/2016/june/project-triforce-run-afl-on-everything/`.

25. HERTZ, Jesse; NEWSHAM, Tim. *TriforceLinuxSyscallFuzzer*. 2016 (accessed 19 November, 2018). Available also from: `https://github.com/nccgroup/TriforceLinuxSyscallFuzzer/blob/master/docs/TestFiles.md`.

26. LECUN, Yann; BENGIO, Yoshua; HINTON, Geoffrey. Deep learning. *nature*. 2015, vol. 521, no. 7553, pp. 436.

27. CHO, Kyunghyun; VAN MERRIËNBOER, Bart; GULCEHRE, Caglar; BAHDANAU, Dzmitry; BOUGARES, Fethi; SCHWENK, Holger; BENGIO, Yoshua. Learning phrase representations using RNN encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*. 2014.

28. SUTSKEVER, Ilya; VINYALS, Oriol; LE, Quoc V. Sequence to sequence learning with neural networks. In: *Advances in neural information processing systems*. 2014, pp. 3104–3112.

29. BOULANGER-LEWANDOWSKI, Nicolas; BENGIO, Yoshua; VINCENT, Pascal. Modeling temporal dependencies in high-dimensional sequences: Application to polyphonic music generation and transcription. *arXiv preprint arXiv:1206.6392*. 2012.

30. NAYEBI, Aran; VITELLI, Matt. Gruv: Algorithmic music generation using recurrent neural networks. *Course CS224D: Deep Learning for Natural Language Processing (Stanford)*. 2015.

31. LIPTON, Zachary C; BERKOWITZ, John; ELKAN, Charles. A critical review of recurrent neural networks for sequence learning. *arXiv preprint arXiv:1506.00019*. 2015.

32. GREGOR, Karol; DANIHELKA, Ivo; GRAVES, Alex; REZENDE, Danilo Jimenez; WIERSTRA, Daan. Draw: A recurrent neural network for image generation. *arXiv preprint arXiv:1502.04623*. 2015.

33. GU, Xiaodong; ZHANG, Hongyu; ZHANG, Dongmei; KIM, Sunghun. Deep API learning. In: *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 2016, pp. 631–642.

34. PAGES, man. *STRACE(1) General Commands Manual*. 2017 (accessed 19 November, 2018). Available also from: `http://man7.org/linux/man-pages/man1/strace.1.html`.

35. DEVELOPERS, LTP. *Testing Linux, one syscall at a time.* 2012 (accessed 19 November, 2018). Available also from: `https://linux-test-project.github.io/`.

36. DEVELOPERS, Linux Kernel. *Linux Kernel Selftests*. 2018 (accessed 19 November, 2018). Available also from: `https://www.kernel.org/doc/Documentation/kselftest.txt`.