2017 2nd International Conference on Artificial Intelligence and Engineering Applications (AIEA 2017)
ISBN: 978-1-60595-485-1

Synthesis of Linux Kernel Fuzzing Tools Based on Syscall

SHUAI BAI, DAN LI, MINHUAN HUANG and HUA CHEN

ABSTRACT

Any software especially the operating system requires testing and evaluation to validate the functional and security characteristics. As yet, fuzzing has become widely adopted into practice software testing. We focus on the operating system kernel fuzzing, select three typical kernel fuzzing tools to analyze. We make synthesis of the three kernel fuzzing tools from the aspects of sisal arguments model, test case construction and fuzzing scheduling and propose an abstract of partial specialization of model and explain it through these tools. Meanwhile, we inspect these tools on the usage of coverage-based fuzzing which is the state-of-the-art fuzzing optimization technology.

KEYWORDS

Fuzzing, Linux kernel, sisal, coverage, genetic algorithm, test case generation.

INTRODUCTION

With the development of the Internet, people have been used to deal with affairs on their phones or laptops. Lots of privacy data have been generated during the use of these devices and stored in the database on the cloud or directly in user's devices. People's privacy data will face very significant security threats once the cloud or user's devices suffer a serious attack. The ensuing massive information security events make people pay more attention to computer system security and personal information security [1]. As an important infrastructure for computer systems, the operating system manages the hardware and software resources. All data in the OS will suffer the risk of leakage if serious vulnerability being found in the OS. So, making vulnerability analysis of operating system and repairing it instantly can reduce the massive security incidents effectively, and make the OS more robust.

Shuai Bai, National Key Laboratory of Science and Technology on Information System Security, Beijing Institute of System Engineering, Beijing 100101, China;
baishuai.io@foxmail.com.
Dan Li, National Key Laboratory of Science and Technology on Information System Security, Beijing Institute of System Engineering, Beijing 100101, China;
yumiko0@mail.ustc.edu.cn
Minhuan Huang and Hua Chen, National Key Laboratory of Science and Technology on Information System Security, Beijing Institute of System Engineering, Beijing 100101, China;

Fuzzing is a prominent software vulnerability analysis technology. It uses automatically generated unexpected or random data as input, invokes program execution automatically and then monitors the execution for exceptions for finding potential vulnerabilities [2]. Fuzzing enjoys great popularity among researchers as more and more people pay attention to computer system security. After several decades of research, fuzzing has been very efficient for testing user-level applications [3, 4], and also formed a pretty universal test and repair flow in the software lifecycle [5]. Fuzzing on the operating system kernel level starts late, but also makes great advance.

In this article, we introduce the latest research progress of vulnerability analysis of operating system kernel based on sisal API. Section 2 gives an overview of the main difficulties and key technologies in fuzzing of OS kernel. Section 3 talks about test case generation method based on API arguments model. Section 4 is about the scheduling optimization and the related tools. Section 5 discusses the design architecture of the system and some experimental results. Finally, section 6 discusses the future work and makes the conclusion.

OVERVIEW

The main feature of fuzzing is the large amount of automatic execution. With the advantage of quantity, it reaches high test coverage. For each special test target, fuzzing needs to do all the test steps automatically, including constructing the testcase, invoking the target to execute and monitoring the exception during execution. In the latest research, coverage-based scheduling can greatly improve the test efficiency [6, 7], and has become a heated research field. As for the programs which run in user-space, there are advanced technologies and tools that can be used during fuzzing, most of which the researchers focus on how to optimize the parameters. But when the fuzzing target changes to the OS kernel, firstly we are suffering how to implement these basic steps automatically. We can further study the optimization only when all the basic steps have been solved.

After an extensive survey, we finally choose three fuzzing tools for Linux kernel and make deeper analysis. We analyze these important steps of the specific implementation. Furthermore, we can do further research about fuzzing optimization based on the understanding of these key technologies. These three tools are Trinity, Syzkaller and TriforceLinuxSyscallFuzzer (TLSF). The target of all the three tools is Linux. On the one hand, Linux is an open source operating system kernel making it easy to obtain rich documents and the source code, modify source code and rebuild the kernel. On the other hand, with Android' share in the mobile market, Linux has become the most popular operating system up to date [8].

As for test case, these three tools also make the same choice, which is sisal. Syscall API is a set of standard interfaces provided by system kernel and user-space program can interact with kernel-space through these sisals. User-space program can access to hardware devices, request system resources, control devices to read or write, create new processes via syscall API. To protect kernel from arbitrary unexpected syscall arguments, there are strict validation about the arguments. Only the test case which passes the validation can be executed. Syscall reflects the functional and technical characteristics of the operating system, and its vulnerability reflects the

vulnerability of the operating system. So, the syscall API is actually a good breakthrough point of vulnerability analysis of the operating system.

In the coverage-based fuzzing, it is necessary to obtain the coverage when the test case is executed in order to implement the optimization based on genetic algorithm. But it's hard to find an easy-to-use tool which can obtain coverage during kernel's runtime. In table 1, we give a brief introduction about these three tools.

TESTCASE CONSTRUCTION BASED ON SYSCALL MODEL

Fuzz tests the target using a large number of automatically constructed testcases as input. Well, a large number of randomly constructed testcases which can't pass the initial validation of syscall arguments can waste a lot of resources. By using a suitable model or input format to constrain the testcase construction, the search space for constructing testcases can reduce a lot and a significantly higher proportion of testcases can pass the arguments validation of syscall. So, it's very useful to improve the efficiency of syscall fuzz.

Model of syscall arguments

The Linux kernel is written in C language, most of the syscall return a status code as idiomatic paradigm in C. Extra information needed by the caller is returned in form of a pre-allocated struct. So, there're lots of structs defined in the kernel to represent this information. To make a fuzzer smarter, the fuzzer needs to know enough information about the syscall arguments, such as the number of the arguments, the type of the arguments, the range of the argument value and so on. Fuzzer can construct better arguments that meet the requirements by leveraging this information.

DESCRIPTION OF SYSCALL ARGUMENT MODEL

Both Trinity and Syzkaller have built a model to describe the Linux syscall, and there're some same contents in the two models. Both models have descriptions about syscall name, number of arguments, type of each argument and type of return value. The type of arguments used in model does not correspond to the type in kernel source code directly, instead it is summarized according to the characteristics of the Linux syscall.

TABLE 1. OVERVIEW OF LINUX KERNEL FUZZING TOOLS.

Project	Brief introduction	Syscall Model	Coverage
trinity	Linux sisal fuzzing tool, has a basic model about sisal arguments.	✓	×
syzkaller	Linux sisal fuzzing tool, has a model about sisal arguments and optimization based on coverage.	✓	✓
TriforceLinux SyscallFuzzer	Derivative based on all, obtain instruction-level coverage via mum trace	×	✓

The type of arguments in Trinity's model is concluded as random_int, fad, len_address, non_null_adress, pid, range, op, list, rampage, cup, pathname, love, ioveclen, socked, sockaddr_len. Each syscall in the kernel is annotated according to these types, and an additional argument sanitize function is designed for each syscall to sanitize the illegal arguments. For different sisals, the annotated types do not have the ability to make a distinction. It's difficult to implement additional types in Trinity. For an address type argument, the annotation can't tell which specific strict that the memory points to and can't allocate the appropriate memory space according to the struct size.

In Syzkaller's model, the type of arguments is concluded as cost, intN, inapt, flags, array, ptr, buffer, string, strconst, filename, len, bytesize, vma, and proc. There's a big difference compared to Trinity that each type in Syzkaller has an optional additional information to enrich its knowledge. For example, const type has additional information that specifies the actual const value, ptr type has additional information that tells the struct where it points to. Furthermore, Syzkaller's model has the grammar to declare new types, fad [int] means the file descriptor consists of an integer or the internal type of the file descriptor is an integer.

DESCRIPTION OF SPECIAL TYPE

There're some types in Linux kernel which can only be generated by sisal, we call these types as resource type.

In Trinity's model, it's restricted to derive new type in the type system, only the concluded type can be used. The most common resource type is included in the type system, and for these resource types in Trinity's type system, there're special construct process to make the resource available.

The Syzkaller's model supports additional type declaration and its use, which enriches the type system implemented in Syzkaller. Resource type can be used in sisal description as long as it has been declared. We call S the constructor of resource type R if the return type of sisal S is R. So we can construct instance of argument type R through its constructor by looking up a constructor table.

PARTIAL SPECIALIZATION OF MODEL

The partial specialization of a model is defined as making partial arguments in a syscall description to fixed values. For the extremely complex syscall IOCtl in Linux, the partial specialization is very useful to split it into a variety of different sisals for different devices and make targeted testing.

In syzkaller, the partial specialization can be easily implemented by only modifying some arguments to const type in syscall description. Partial specialization in Trinity needs additional syscall description and corresponding sanitize function.

Testcase construction

The syscall description with rich argument information can be obtained by modeling the syscall and describing it based on the model. And the test case with specific constraints can be further constructed. The method of constructing test case in fuzzing can be divided into two kinds, generation-based and mutation-based. Trinity use the generation-based method while Syzkaller combines the two and forms a mixed

method. TLSF does not have the grammar knowledge about syscall, so it can only construct testcase by mutation.

Manifestation of testcase

Although they are all Linux kernel fuzzing tools based on syscall API, the manifestation of testcases still has a big difference. In Trinity, a syscall with arguments is a testcase. Trinity generates testcases and invokes to execution continuously until a system crash occurs. In Syzkaller, a testcase consists of a sequence of syscalls, which is closer to the execution in reality, and can bring the dependence of arguments in context. This form of testcase makes the testcase generation method more diversified.

GENERATION-BASED TESTCASE CONSTRUCTION

In Trinity, the generation-based testcase construction method is used. The model description of the syscall represents the grammar knowledge of the testcase. For each argument, the value is generated based on the annotated type information. At the beginning of Trinity execution, a large number of resource type values are initialized and saved to a resource pool. When a resource type is required by a syscall, it randomly choose an instance value of corresponding resource type. Another important way to ensure the validity of the arguments is the sanitize function. Sanitize function filters the error arguments or modifies error arguments to the default value. Also sanitize function do some special generation for the type that the type system does not handle.

MIXED TESTCASE CONSTRUCTION

Syzkaller combines generation-based and mutation-based test case construction method, it generates testcase using the grammar constraints in the model description or selects a testcase from the testcase corpus and mutates from the selected one.

In Syzkaller, a testcase consists of a sequence of syscalls. In order to solve the dependences of syscall arguments in the context, Syzkaller maintains a choice table during generating a testcase. The choice table contains information such as values of resource and strings that have been generated in the prior generation process. When meets the same argument type in later generation process, Syzkaller tries to reuse the existing argument values in the choice table with a certain probability, or it generates a new one. The whole process of Syzkaller's test case generation is described in algorithm 1.

Algorithm 1 generate testcase a testcase which has n syscalls

```
1: while the number of syscall in testcase t is less than n
2:     choose a syscall s from syscall list
3:     for each argument a in s:
4:     if the annotated type of a is resource type:
5:         reuse the existing values in the choice table with a certain probability
6:         or look up the constructor of the resource type and generate it recursively
7:     else
8:         generate value according the annotated type
9:     end for
10:    append s to t
11:    update choice table
12:end while
```

There're also mutations in the granularity of one syscall in the test cases construction of Syzkaller. For example, insert a new syscall to the test case, or insert a syscall from another testcase in the corpus, or delete a syscall from a test case.

SCHEDULING OPTIMIZATION OF TESTING PROCESS

Both TLSF and Syzkaller have adopted coverage-guided scheduling optimization of the fuzz testing. To be more specific, the scheduling optimization process is conducted by a modified genetic algorithm, which was firstly used in the AFL. In order to achieve coverage-guided scheduling optimization, it is essential to obtain the coverage during the kernel runtime. However, the methods for obtaining the coverage taken by TLSF and Syzkaller are vastly different. Syzkaller directly gets the coverage from the interior of the system, while TLSF implements the same function with the aid of virtual machine management tools.

Obtain Kernel Coverage based on KCOV

KCOV is a specially developed kernel tool which can provide code coverage collection of the kernel for coverage-guided fuzzing. The coverage is collected by static kernel instrumentation. KCOV modifies the process basic block task_struct and stores the function return address from inside the instrumented function. The return address which represents the address of the next instruction that should be executed indicates all the branch paths that the kernel has passed during the execution. Meanwhile, KCOV skillfully realizes the transmission process of the coverage between the kernel-space and the user-space via using debugfs, a virtual file system for kernel debugging.

Obtain Kernel Coverage based on Virtual Machine

TLSF uses qemu trace to follow the tracks of the guest operating system. By adding hypercall [9], TLSF can realize the communication between the QEMU guest operating system and the virtual machine hypervisor, as well as the control of the address segment which is being followed.

In order to obtain the coverage from the interior of the system, it is necessary to revise the kernel source and be clear about its running state, which also contributes to the control of the position of the instrumented point. Collecting the coverage from inside the system can follow the tracks of multi-process coverage independently. However, to gain the coverage by means of virtual machine hypervisor, modifying the QEMU source code and creating a new hypercall seems to be essential. In this way, it becomes possible to control the starting point and the ending point of a tracing process. For the virtual machine hypervisor, the operating system is only regarded as a black-box. There is no concept of internal process so that obtaining multi-process coverage seems to be impossible.

Scheduling Optimization of Fuzz Testing Using Genetic Algorithm

As stated above, TLSF and Syzkaller adopted different methods to collect the coverage. Nevertheless, the processes taken by the two test tools of scheduling

optimization of fuzzing test using the coverage are quite similar. Both of them choose to optimize the system via genetic algorithm. Testcase corresponds to the individual in the genetic algorithm and in the meanwhile testcase corpus is equivalent to the population. When newly generated testcase returns higher-coverage finds, the testcase will be added to the corpus with the update on its coverage.

Genetic algorithm is inspired by the process of natural selection so that it simulates the derivation law of the biotic population. To generate a second-generation population, genetic algorithm makes use of a combination of genetic operators: crossover (also called recombination), and mutation. Solutions are selected through a fitness-based process, where fitter solutions which are measured by a fitness function are more likely to be selected. The selection methods rate the fitness of each solution to search for the best solutions. The generational process is repeated and eventually helps the population to grow until ranking solution's fitness is reaching [10].

The fitness functions chosen by Syzkaller and TLSF are quite the same. They both filter the next generated individuals by judging whether the teat case has triggered new paths. Based on the length of the coverage of the testcases, they sort the testcase corpus and select fitter solutions to control the population size.

IMPLEMENTATION SCHEME AND SYSTEM COMPARISON

System Structure Design

The three testing tools vary a lot in system composition design aspect due to their diversities in technical schemes during the fuzz testing process.

When the system crashes, it can neither recover from the crashes nor store the crash data automatically. TLSF and Syzkaller both take advantage of virtual machines to insulate the target test system from the host system. With the aid of the virtual machine hypervisor, the two tools implement the fault recovery mechanism as well as the storage of crash data. TLSF runs a driver in the user mode inside the virtual machine and communicates with the hypervisor by using hyper call. Test cases are generated by AFL fuzzier outside the virtual machine, which are then being written to the specific memory addresses inside the virtual machine via the hypervisor. The internal driver analyzes and executes the test cases from the specific address areas. It also takes control of the trace switches of the hypervisor by hyper call. Coverage can be directly collected using mum trace.

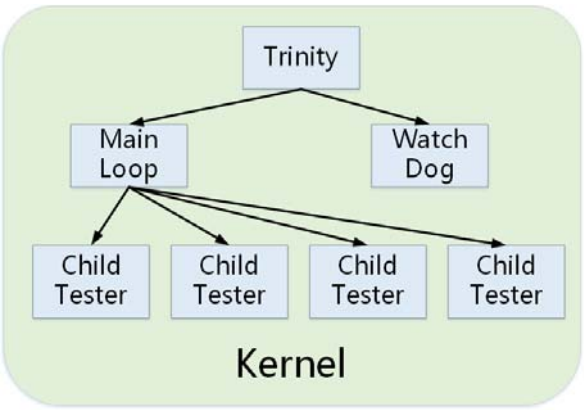

Figure 1. System Structure of Trinity.

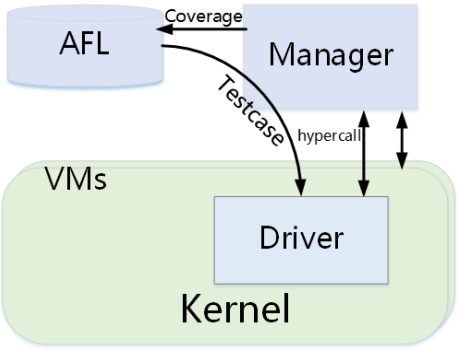

Figure 2. System Structure of TLSFFirst of all, the system structure taken by TLSF is the easiest to understand but it also results in the weakest degree of automation.

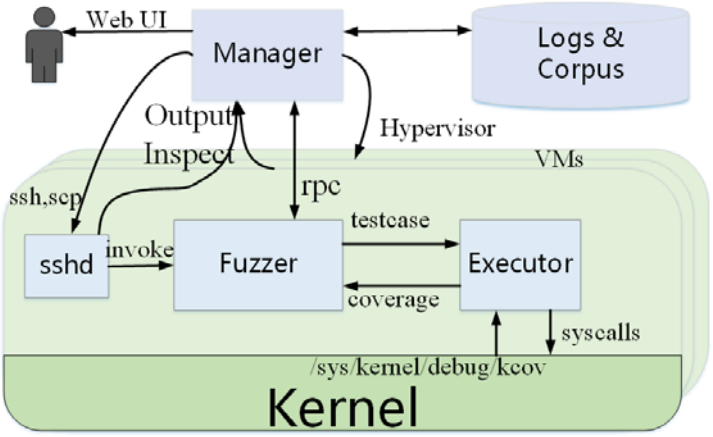

Figure 3. System Structure of Syzkaller.

Nevertheless, the way that test cases are generated and coverage are collected are different in Syzkaller. These two processes are both executed inside the virtual machine. The internal processes generate and execute the test cases, collect coverage via the internal KCOV tool and update the test case corpus, with timing synchronization with the hypervisor including the running state of the corpus and so forth.

Error Handling Mechanism

The goal of fuzz testing is to discover the vulnerabilities in the system and analyze the exception information. As a consequence, it is essential to take into account the error recurrence mechanism when designing the fuzz testing tools. All of the three tools that we have discussed about have realized varying degrees of error handling mechanism.

Trinity did not make use of virtual machine when it was designed, so it lacks of the capability to recover from failure and its testing process should be supervised by manpower. Every time a new test case is generated, Trinity will store it before executing it. If the fuzz testing process crashes, the system will automatically keep an account of the crash file so that the tester can reproduce the crash and locate the unknown exception in the kernel after the system's rebooting by using the crash file and the lastly-stored test case. TLSF controls the recovery of failure with the aid of hypervisor. It orients the logging of the virtual machine to the exterior of the system.

As stated before, the test cases are also generated outside the system, making it possible to reproduce and locate the error location using the latest test case and the logging information when the virtual machine crashes. Syzkaller provides the ability of error reproduction by reproducing the output serialized test cases before they are being executed. By monitoring the logging output, the log snippet and the corresponding test case are stored once an error occurs. In the meanwhile, Syzkaller starts the error reproduction process to re-execute the stored test case and judge automatically whether the crash occurs again. To sum up, Syzkaller realizes a sustained and fully automated testing process.

CONCLUSION

This article makes a detailed comparison between three typical testing tools specifically designed for Linux kernel fuzz testing. To draw a conclusion, Syzkaller is the most outstanding open-source fuzzier for operating system kernel fuzz testing up to date based on the generation of test case, the scheduling optimization of testing process, the automation degree as well as the error handling mechanism. From the arguments modeling aspect, Syzkaller can provide more specific and easier-to-extend arguments models than Trinity with higher efficiency in test case generation. As for the scheduling optimization of testing process, Syzkaller realizes coverage collecting from the interior of the system as well as the coverage-guided scheduling optimization algorithm. Compared to TLSF, its coverage can be collected by multi-processes running independently, which highly increases the efficiency of the fuzzing process. When it comes to system integration, Syzkaller also implements the function of a sustained and unsupervised testing process fully automatically. Speaking of feedback optimization of fuzz testing, coverage-guided genetic algorithm stays the mainstream direction of correlation studies at present, which should be further investigated of to improve the efficiency of test case fitness evaluation methods.

REFERENCES

1. L. Ablon and A. Bogart, Zero Days, Thousands of Nights: The Life and Times of Zero-Day Vulnerabilities and Their Exploits. Rand Corporation, 2017.
2. M. Sutton, A. Greene, and P. Amina, Fuzzing: brute force vulnerability discovery. Pearson Education, 2007.
3. V.-T. Pham, M. Böhme, and A. Roychoudhury, “Model-Based White box Fuzzing for Program Binaries,” Proc. 31st IEEEACM Int. Conf. Autom. Softw. Eng., pp. 552–562, 2016.
4. S.K. Cha, M. Woo, and D. Brumley, “Program-adaptive mutational fuzzing,” in Security and Privacy (SP), 2015 IEEE Symposium on, 2015, vol. 2015–July, pp. 725–741.
5. S. Mansfield-Devine, “Open source software: determining the real risk posed by vulnerabilities,” Netw. Secur., vol. 2017, no. 1, pp. 7–12, 2017.
6. M. Böhme, V.-T. Pham, and A. Roy Choudhury, “Coverage-based Greyson Fuzzing as Markov Chain,” Proc. 23rd ACM Conf. Compute. Common. Secure. pp. 1–12, 2016.
7. K. Serebryany, “Continuous Fuzzing with libFuzzer and Address Sanitizer,” in Cybersecurity Development (SecDev), IEEE, 2016, pp. 157–157.
8. “W3Counter: Global Web Stats.” Available: https://www.w3counter.com/globalstats.php.
9. J. Nakajima and A.K. Mallick, “Hybrid-virtualization—enhanced virtualization for Linux,” in Proceedings of the Linux Symposium, 2007, vol. 2, pp. 87–96.
10. E. Jääskelä, “Genetic Algorithm in Code Coverage Guided Fuzz Testing,” University of Oulu, 2015.