A Review on Library Fuzzing Tools

Jia Song *jsong@uidaho.edu*
Computer Science Department
University of Idaho
Moscow, Idaho, 83844, USA

Abstract

Fuzzing is a powerful software security testing technique. It can be automated and can test programs with many randomly generated fuzzing inputs to trigger overlooked bugs. Libraries and functions are commonly used by programmers to be directly called from their programs. However, most programmers would simply use public libraries without doubting whether these libraries are secure or not.To help with it, library fuzzing has been proposed.Fuzzing a whole program is very common, however, fuzzing a standalone function or library is challenging. Different from an executable program, functions cannot be run on themselves. In addition, randomly generating certain parameters might break the relationships between parameters and therefore result in a large number of false positives.There has been not much research in the area of library fuzzing. However, library or function fuzzing could be a very useful testing toolfor programmers and developers. This paper reviews the recent research work related to library fuzzing and function fuzzing. The results may be helpful to any researchers who plan to explore this research area.

Keywords: Fuzzing, Library Fuzzing, Software Testing, Function Fuzzing.

1. INTRODUCTION

Fuzzing is one of the software security testing methods. It generates random input which can be fed to a fuzzer to test programs. The testing process is monitored to specifically look for program crashes. The advantages of fuzz testing include that it can be automated and does not require access to the source code. It can quickly generate malformed fuzzing inputs and feeds them into the program under test. Then the system will monitor any raised exceptions or failures while running the program using these fuzzing inputs. Programmers tend to expect users to always provide valid input as they interact with the program.However, attackers always give unexpected input to try to mess up the program execution. In addition, people make mistakes accidentally by entering invalid input into the program. Fuzz testing is extremely useful for testing input components of programs because the randomly generated input could test overlooked cases in the programs.

In traditional fuzzing, because the randomly generated input usually fails to satisfy the required format of the input, a well-known drawback of traditional fuzz testing is that it is ineffective at triggering bugs at the deeper level of the program (Sutton et al., 2007). Most of the programs which require user interaction expect user input to be in a certain format. For example, a command followed by zero or more parameters. If the input does not match the required input format, then the input may be rejected directly by the program. Since traditional fuzzing generates fuzzing input randomly, a large portion of the fuzzing input may fail to satisfy the input format requirement at the early stage of the program execution. Therefore, the fuzzing input may be dropped before reaching deeper levels of the code, hence leaving the deeper code unreachable and untested.

Different from fuzzing regular programs, library fuzzing focuses on fuzzing a single function or multiple functions in a library. Library fuzzing becomes a hot research topic because of the security concerns of existing libraries. Libraries are commonly used by programmers and

software developers to reduce their workload on coding. For example, modern operating systems, such as Ubuntu and Debian, consistof a base system and hundreds of libraries and services (Ispoglou et al., 2020). These libraries are a handful to software developers because they can be directly used by the developers without the need of understanding the internal details. Similarly, no matter of writing code in C/C++ or Python, different libraries are frequently used by programmers in their programs without many inspections of the security of those libraries. As software security becomes an important issue, more and more developers started to question the security of libraries. Many libraries are existing there, but a lot of them have not been examined or tested for their security. To help test the security of libraries, library fuzzing can be a good solution. This paper focuses on reviewing the existing fuzzers which test standalone libraries. The result may be helpful to any researchers who are considering conducting research in the library fuzzing area.

2. BACKGROUND

Fuzzing is a promising technique to discover vulnerabilities in software automatically. The traditional fuzzing tool, proposed by Miller et al. (Miller et al., 1990) in 1990, is a form of black-box testing. It uses a fuzzing tool to randomly generate or mutate input strings. Miller et al. indicated that their fuzzing tool successfully crashed 25% to 33% of UNIX utility programs. Since then, many studies (Miller et al., 1990, Sutton et al., 2007, Oehlert, 2005) have proven fuzzing to be surprisingly effective in revealing vulnerabilities in software systems.

Based on the knowledge of the target program, fuzzing can be categorized into three groups: white-box fuzzing, grey-box fuzzing, and black-box fuzzing (Manes et al., 2019). With more information on the testing program, white-box fuzzing could be more effective in revealing bugs and can be used to target a specific portion of the program. Grey-box fuzzing has certain knowledge of the target program, and it usually collects information through static analysis and/or dynamic analysis of the program to improve the effectiveness of fuzzing. Black-box fuzzing has no information about the program, such as no access to source code, and therefore can only observe the input to the target program and output from the execution of the program. However black-box fuzzing is easy to set up.

To make fuzzing smarter, researchers have combined different techniques, such as symbolic or concolic execution, taint analysis, and grammar detection, with fuzzing to help direct the testing. In symbolic execution, symbolic values are used instead of concrete values as input values, and symbolic expressions are used to represent the values of program variables (Cadar and Sen, 2013). When testing software, the symbolic execution technique is used especiallyto explore as many execution paths as possible. Fuzzing tools guided by symbolic or concolic execution (Yun et al., 2018, Cadar et al., 2011, Mouzarani et al., 2015, Ognawala et al., 2017, Stephens et al., 2016), are good at exploring program paths. A major problem with symbolic execution is the path explosion problem, which leadsto the symbolic execution technique requiringa large amount of computing power and running overhead.

With taint analysis (Suh et al., 2004),all of the data received from untrusted sources can be marked as tainted. These tags, representing whether a tagged data is tainted or untainted, are propagated while the program's execution. The tags are checked when running certain operations, such as making security-related decisions. If the tainted data are used, either further checks will be done, or the tainted data is simply rejected and not used. Fuzzers assisted by taint analysis(Ganesh et al., 2009, Bekrar et al., 2012, Cai et al., 2014, Liang et al., 2022) are more effective when detecting vulnerabilities caused by input from unsafe sources, but the processes of propagating and checking tags add extra execution time.

Grammar-guided fuzzing (Hoschele and Zeller, 2016, Hoschele et al., 2017, Bastani et al., 2017, Godefroid et al., 2017, Salem and Song, 2021)is another method to improve the quality of fuzzing input. Some of the grammar-based fuzzers include extra work from the tester, who need to enter the accepted grammar or format of the valid input to the fuzzer. Others may extract grammars

from sample valid input files to the program, and then use the extracted grammars to help generate effective fuzzing input. To keep track of the coverage of the test cases, code coverage information isusuallymaintained (Lou and Song, 2020).

3. DIFFICULTIES IN LIBRARY FUZZING

In fuzzing, a large amount of input is fed into the target program and the output from the execution of the program is monitored. However, what makes library fuzzing difficult is the fact that libraries cannot be executed as standalone programs. Instead, functions in libraries are called at different places of the programs with the required arguments. Therefore, in practice, library fuzzing (eg.libFuzzer) usually requires the help of an analyst to manually write a fuzzing stubto call functions in the target library (Manes et al., 2019). The fuzzing stub can call the target library functions with the required arguments and later it can be leveraged with random arguments to fuzz the target function.

Fuzzing functions with random arguments are less likely to produce efficient fuzzing (Ispoglou et al., 2020). For example, a function that manipulates an integer array usually gets two arguments.One of themstores the start address of the array and the other one indicates the length of the array. Library fuzzing may not understand the relationship between the two parameters. Hence feeding the function with random fuzzing input may cause a large number of false positive crashes. Therefore, to develop a good fuzzer stub, the analyst who writes it needs to have a good understanding of the target library and the possible relationship among the arguments.

4. LITERATURE REVIEW

Fuzzing kernel code or device drivers is a challenging task. For example, kernel and device drivers use pointers and function pointer tables frequently, however, the random value generated by the fuzzer cannot satisfy the requirement in most cases. In addition, it is difficult to figure out the right sequence and proper arguments of system calls to be able to trigger the bugs located deeply in the system (Kim et al., 2020). On the other hand, one good thing about kernel fuzzing is that most of the system kernels and device drivers are open sources (Corina et al., 2017), so the source code can be analyzed to extract important information to guide fuzzers. To make kernel fuzzers more effective, different information about the target is used to guide fuzzers. For instance, Trinity (Jones, 2011), Syzkaller (Syzkaller, 2017), and DIFUZE (Corina et al., 2017) are type-aware kernel fuzzers that use data type information derived from the system calls. Syzkaller requires an analyst to provide the information while Trinity and DIFUZE can analyze the target code and generate this type of information automatically. Code coverage feedback and symbolic or concolic execution are also common techniques used to guide fuzzers. For example, Syzkaller (Syzkaller, 2017), kernel-fuzzing (Kernel-fuzzing, 2016), and HFL (Kim et al., 2020) leverage code coverage feedback to guide generations of the test cases where HFL also employs concolic execution technique.

4.1 System Call and Device Driver Fuzzing

Trinity (Jones, 2011) is a type-aware kernel fuzzer. It generates fuzzing input using the data type information from system call prototype definitions. When fuzzing a function, Trinity learns the data type of each parameter in that function and randomly picks or generates a value to test it. For example, when testing a *read* system call, the first parameter requires a file descriptor. Trinity maintains a list of valid file descriptors and randomly picks one of them to pass to the *read* system call. When encounters a length parameter, it randomly generates an integer value to be used as the length parameter. Given the fact that there will be a lot of different combinations for parameters of a system call, Trinity does not test everything (Jones, 2011).

Syzkaller (Syzkaller, 2017) was developed to fuzz Linux system calls and later extended to work with other OS kernels. It is a widely used tool adopted by researchers to fuzz kernel code. It requires analysts to manually provide information about the target system call, such as arguments to the call, and then fuzzes the system calls. Syzkaller consists of three main

components, syz-manager, syz-fuzzer, and syz-executor. syz-manager interacts with the VM environment from outside of the VMs. And syz-fuzzer and syz-executor are running inside VMs, with syz-fuzzer focusing on generating fuzzing input and syz-executor executing the target system call with the input from syz-fuzzer (Li and Chen, 2019). Syzkaller is a grey-box fuzzer and it is typically used on kernels compiled with kernel code coverage feedback, such as from kcov (kcov, 2011).

Implemented by Han and Cha, IMF (Han and Cha, 2017) is a fuzzing system that targets finding latent bugs located deeply in the kernel code. The fuzzer employs a model-based approach that deduces an API model from executing the regular program, then the model is used to help generate fuzzing input. An API model indicates two types of dependences among function calls: (1) an ordering dependence which indicates the order of function calls, and (2) a value dependence which keeps track of the relationship between an output from a function to an input of another function (eg. a return value from a function is used as an argument to another function) (Han and Cha, 2017). IMF consists of three components, a logger, an inferrer, and a fuzzer. The goal of the logger is to execute the program with a given set of inputs and log everything, such as the parameter in the target function, and the return values. Then the inferrer component analyzes the API logs and figures out the API model which specifies the ordering and value dependences among function calls. This API model is sent to the fuzzer component to generate fuzzing inputs which are executable programs. These programs are executed by the fuzzer and IMF monitors whether a crash happened or not.

DIFUZE (Corina et al., 2017) is an interface-aware fuzzing tool that generates valid inputs to execute the kernel drivers in modern Unix-like systems. It focuses on the exploration of the *Ioctl* interface provided by different device drivers. *Ioctl* is a device-specific system call that allows for input and output operations to be processed by a device driver. For every driver, DIFUZE performs static analysis on the driver code to identify all the *Ioctl* entry points and the corresponding structures (eg. types of arguments). Corina et al. have integrated DIFUZE into syzkaller (Corina et al., 2017). DIFUZE can convert the results of the analysis into a format required by syzkaller. Then syzkaller performs fuzzing on the system call. A weakness of DIFUZE is that if the driver crashes at the beginning, the deeper driver code will not be fuzzed because of the earlier crash. In addition, the authors indicate that DIFUZE is unable to analyze complex relationships between arguments (Corina et al., 2017). Similar to DIFUZE, other fuzzing tools, iofuzz (ioctl, 2014) and ioctlfuzzer (ioctl, 2011) can test the *Ioctl* interfaces for Windows kernels.

HFL (Kim et al., 2020) combines traditional fuzzing and concolic execution to fuzz Linux system calls. HFL maintains a template which is a list of existing system calls and the corresponding structures. This includes the type of parameters in system calls, a range of constant values of certain parameters, the relationships between systems calls, etc. (Kim et al., 2020). Based on these pre-defined system call templates, HFL can generate or mutate programs that execute system calls in certain orders with different parameters. HFL instruments all blocks of code in the kernel, so that fuzzing can be guided by the coverage feedback. Its goal is to mutate fuzzing input to reach a bigger execution coverage. A frequency table is maintained by HFL during fuzzing to keep track of whether a branch is hard-branch (a branch is always evaluated to True or False) or not. When a hard branch is identified by fuzzing, it will be sent to the symbolic analyzer to help find out a value that can flip the evaluation of the hard branch (Kim et al., 2020). HFL obtains potential dependency pairs from the results of static analysis on the kernel code. It then performs runtime validation on the dependency data to identify true dependency pairs. By doing this, HFL achieves better efficiency and effectiveness in fuzzing kernel space (Kim et al., 2020).

4.2 Function and Library Fuzzing

A challenge of function fuzzing is function cannot be executed as a standalone program. Therefore, to fuzz a function, the fuzzer needs to have a proper way to call the target function and pass a correct number of arguments to the function. Some function fuzzers require analysts to write fuzzing stubs manually to call the target function (libFuzzer, 2016). This not only requires analysts to have a good understanding of the target but adds extra workload to the analysts. To

solve this problem, several advanced function fuzzers can analyze the program or source code to derive dependence information among functions to generate fuzzing stubs automatically (Ispoglou et al., 2020, Blair et al., 2020). To fuzz only a portion of a program, the idea of in-memory fuzzing was proposed (Sutton et al., 2007, Hoglund, 2003) The advantage of in-memory fuzzing is that it can reduce the execution overhead by eliminating the process of re-executing the whole program and only test a small portion of the program (Manes et al., 2019). Several fuzzers that implemented this idea include GRR (Trail of Bits, 2016), AFL in persistent mode (Zalewski, 2015), IMF-SIM (Wang and Wu, 2017), etc.

LibFuzzeris a coverage-guided fuzzing engine.It can be used to test libraries or a single function (libFuzzer, 2016). Supported by the LLVM's built-inSanitizerCoverage instrumentation (SAN, 2017), LibFuzzer gets code coverage feedback to detect which parts of the target code are reached and which are not. Then it mutates the input data to try to reach that code area. LibFuzzer can work without any input data but will not be efficient especially when the target functions require complex or structured input. The weakness of LibFuzzer is it requires source code and some human involvement to write fuzzing stub which is used to call the target function. Several advanced fuzzer use LibFuzzer to support fuzzing libraries or functions, such as SlowFuzz (Petsios et al., 2017), FuzzGen (Ispoglou et al., 2020), DeepState (Goodman and Groce, 2018).

FuzzGen (Ispoglou et al., 2020), developed by Ispoglou et al., is a fuzzing tool that analyzes the whole system automatically and generates fuzzing stubs which can be fed to the LibFuzzer. A library function may be used multiple times in a program at different places. FuzzGen analyzes the whole program, collects information about each use of the target function, and defines an Abstract API Dependence Graph to keep track of the possible library dependence. The Abstract API Dependence Graph includes control dependencies and data dependencies. The control dependencies tell how the different API calls should be invoked, while data dependencies indicate the relationships of data values, such as an output of a function being used as the input to another function (Ispoglou et al., 2020). Then the dependence graph is used by FuzzGen to generate proper fuzzing stubs for different API calls. The output of FuzzGen is a set of C++ source files and each one is a fuzzer stub. In this way, the fuzzing stubs can be generated without the help of analysts, and they will be fed to the code coverage guided LibFuzzer to perform the actual library fuzzing work (Ispoglou et al., 2020).

Blair et al. proposed a framework, HotFuzz (Blair et al., 2020), which automatically discovers algorithmic complexity vulnerabilities in Java libraries. Algorithmic complexity vulnerability is defined as a situation that a small adversarial input can lead to worst-case behavior (eg. denial-of-service) when processing it. HotFuzz consists of two phases: (1) micro-fuzzing and (2) witness synthesis and validation (Blair et al., 2020). In the micro-fuzzing phase, HotFuzz takes the whole Java program or library as input and tries to automatically generate a test harness for every function in the input. A test harness is a function input that can be fed to the function under test. Then micro-fuzzing invokes the target function with the specific test harness and measures the resources consumption by the target function. If the consumption exceeds a pre-defined threshold, the test harness is sent to the second phase. The goal of the second phase is to validate whether or not the forwarded test harness can produce abnormal resource consumption when testing in a real Java run time environment. If yes, the test case is flagged to indicate an existing vulnerability in the program, otherwise, the test case is discarded to reduce the false positive rate (Blair et al., 2020). The advantage of HotFuzz is that it can be executed without any help from analysts. Several similar fuzzers which generate input to reveal algorithmic complexity vulnerabilities include Slowfuzz(Petsios et al., 2017) and PerfFuzz (Lemieux et al., 2018). Built on top of libFuzzer, Slowfuzz uses an evolutionary guidance engine to generate inputs that can lead to worst-case resource consumption. PerfFuzz is based on AFL, and it implements a performance map and focuses on generating inputs with a higher number of execution paths.

Developed by Patrice Godefroid, MicroX (Godefroid, 2014) is a prototype VM which allows micro execution of binary code. Micro execution means the ability to execute a portion of a binary file

without the user providing input data or test drivers (Godefroid, 2014). In MicroX, a user can simply identify a function or a code location in an *exe* file or *dll* file. Then the VM starts executing the code from that specific location, feeds input value, and catches all memory operations. The value of the input depends on which input mode the VM is running. The VM has several input modes -- zero mode, random mode, file mode, process-dump mode, and SAGE mode (Godefroid, 2014). Each mode indicates a method to generate input, for instance, all input values are set to zeros in zero mode, and 32-bit random values are used as input in random mode. In SAGE mode, a white box fuzzer, SAGE (Godefroid et al., 2012), is used in MicroX to generate input values to fuzz the target code fragment. SAGE can help symbolically execute the code path taken by the micro execution and generate a path constraint for the specific micro execution. Therefore, it can guide MicroX to reach more program paths (Godefroid, 2014).

AFL (Zalewski, 2013) is a popular fuzzer that is widely used by developers and researchers. When fuzzing, AFL starts a new process, feeds in an input, and monitors whether a new path is reached or not. If a new path is reached by using a specific input, AFL puts the input into its queue and later uses it to reach other parts that are deep in the program. AFL constantly creates new processes and tests input and discards them when the input is tested. To avoid the overhead, AFL has an optional persistent mode that can be used (Zalewski, 2015). In persistent mode, instead of using the *execve()*syscall and the linking process, AFL uses the *fork()* call to test each fuzzing input. It performs in-memory fuzzing in a loop, hence does not need to restart the program again and again. The author indicates the speed gains by using the AFL persistent mode can be as high as ten times of the regular AFL. However, there may be problems such as possible accidental memory leaks and Denial of Service conditions in the fuzzing process (Zalewski, 2015).

Proposed by Wang and Wu, IMF-SIM (Wang and Wu, 2017) is a tool for analyzing binary code similarity. The tool leverages in-memory fuzzing to quickly test functions and collects multiple behavior traces. The in-memory fuzzing module of IMF-SIM employs a dynamic binary instrumentation tool, Pin (Luk et al., 2005). Hence an advantage of IMF-SIM is that it does not require access to source code and even works with stripped binaries thathave no debug information or program relocation information. In-memory fuzzing can start executing from any point of a program. When performing in-memory fuzzing, IMF-SIM sets a starting point, then fuzzes from that point with a fuzzing input. After one iteration of fuzzing, the target program is tested from the same starting point with other mutated fuzzing inputs. IMF-SIM suffers from the same problem as other function fuzzing techniques -- no data type information is available. To address this issue, the authors proposed a backward taint analysis method to reveal the root of a pointer data flow (Wang and Wu, 2017). Whenever a pointer dereference error happens, the pointer is tainted and propagated backward until finding the source of the pointer is, such as from a function parameter. In this case, the source is fed with a valid pointer on the later fuzzing iterations.

5. CONCLUSION

Fuzzing a whole program has been studied by many researchers, but there are only a few research works related to fuzzing a standalone library.This paper surveys the existing library fuzzing tools. The target for library fuzzers is usually a function. The advantage of this is that it is less likely to run into the path explosion problem because the number of execution paths of a function is much smaller compared to a whole program. In addition, testing a single function is much faster, which means hundreds and thousands of fuzzing inputs can be tested in a very short time. On the other hand, compared to fuzzing a whole executable, effectively fuzzing a standalone library tend to be much harder. Firstly, most of the library functions need arguments to be passed to the function. If there are relations between two or more arguments, such as an array and its size or a buffer and its size, a library fuzzer may be ineffective because of not knowing this relationship between arguments. Secondly, a function alone is not executable, hence requiring certain ways to call the function and pass the correct number of arguments to it. This step usually requires some help from human testers. Thirdly, to make library fuzzers more efficient, information about the function need to be provided to the fuzzer by human testers. In short, the

research about fuzzing standalone libraries has been explored by some researchers, but more advanced researchcan be conducted to make library fuzzer more effective.

6. ACKNOWLEDGMENT

This research work wassupported through the INL Laboratory Directed Research& Development (LDRD) Program under DOE Idaho Operations Office Contract DE-AC07-05ID14517.

7. REFERENCES

Bastani, O., Sharma, R., Aiken, A., and Liang, P. (2017). Synthesizing program input grammars. In Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, pages 95–110, New York, NY, USA. ACM.

Bekrar, S., Bekrar, C., Groz, R., and Mounier, L. (2012). A taint based approach for smart fuzzing. In 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation, pages 818–825.

Blair, W., Mambretti, A., Arshad, S., Weissbacher, M., Robertson, W., Kirda, E., and Egele, M. (2020). Hotfuzz: Discovering algorithmic denial-of-service vulnerabilities through guided micro-fuzzing. Proceedings 2020 Network and Distributed System Security Symposium.

Cadar, C., Godefroid, P., Khurshid, S., Pasareanu, C. S., Sen, K., Tillmann, N., and Visser, W. (2011). Symbolic execution for software testing in practice: Preliminary assessment. In Proceedings of the 33rd International Conference on Software Engineering, ICSE '11, pages 1066–1071, New York, NY, USA. ACM.

Cadar, C. and Sen, K. (2013). Symbolic execution for software testing: Three decades later. Commun. ACM, 56(2):82–90.

Cai, J., Yang, S., Men, J., and He, J. (2014). Automatic software vulnerability detection based on guided deep fuzzing. In Software Engineering and Service Science (ICSESS), 2014 5th IEEE International Conference on, pages 231–234.

Corina, J., Machiry, A., Salls, C., Shoshitaishvili, Y., Hao, S., Kruegel, C., and Vigna, G. (2017). Difuze: Interface aware fuzzing for kernel drivers. pages 2123–2138.

Cr4shloctl. (2011). Ioctl fuzzer - windows kernel drivers fuzzer. https://github.com/Cr4sh/ioctlfuzzer.

Ganesh, V., Leek, T., and Rinard, M. (2009). Taint-based directed whitebox fuzzing. In Proceedings of the 31st International Conference on Software Engineering, ICSE '09, pages 474–484, Washington, DC, USA. IEEE Computer Society.

Godefroid, P. (2014). Micro execution. In Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pages 539–549, New York, NY, USA. Association for Computing Machinery.

Godefroid, P., Levin, M. Y., and Molnar, D. (2012). Sage: Whitebox fuzzing for security testing. Queue, 10(1):20:20–20:27.

Godefroid, P., Peleg, H., and Singh, R. (2017). Learn&fuzz: Machine learning for input fuzzing. In Proceedings of the 32Nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, pages 50–59, Piscataway, NJ, USA. IEEE Press.

Goodman, P. and Groce, A. (2018). Deepstate: Symbolic unit testing for c and c++.

Han, H. and Cha, S. K. (2017). Imf: Inferred model-based fuzzer. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS '17, pages 2345–2358, New York, NY, USA. Association for Computing Machinery.

Hoglund, G. (2003). Runtime decompilation the 'greybox' process for exploiting software. https://www.blackhat.com/presentations/bh-federal-03/bh-fed-03- hoglund.pdf.

Hoschele, M., Kampmann, A., and Zeller, A. (2017). Active learning of input grammars. CoRR, abs/1708.08731.

Hoschele, M. and Zeller, A. (2016). Mining input grammars from dynamic taints. In Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering, ASE 2016, pages 720–725, New York, NY, USA. ACM.

Jones, D. (2011). Trinity: Linux system call fuzzer. https://github.com/Cr4sh/ioctlfuzzer.

Kcov. (2011). kcov - code coverage analysis for compiled programs and python scripts. https://manpages.debian.org/unstable/kcov/kcov.1.en.html.

Kernel-fuzzing. (2016). Kernel-fuzzing. https://github.com/oracle/kernel- fuzzing.

Kim, K., Jeong, D., Kim, C. H., Jang, Y., Shin, I., and Lee, B. (2020). Hfl: Hybrid fuzzing on the linux kernel.

Libfuzzer. (2016)LibFuzzer- a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html.

Ioctl. (2014). A mutation based user mode (ring3) dumb in-memory windowskernel (ioctl) fuzzer. https://github.com/debasishm89/iofuzz.

Ispoglou, K., Austin, D., Mohan, V., and Payer, M. (2020). Fuzzgen: Automatic fuzzer generation. In 29th USENIX Security Symposium (USENIX Security 20), Boston, MA. USENIX Association.

Lemieux, C., Padhye, R., Sen, K., and Song, D. (2018). Perffuzz: Automatically generating pathological inputs. In Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, pages 254–265, New York, NY, USA. Association for Computing Machinery.

Li, D. and Chen, H. (2019). Fastsyzkaller: Improving fuzz efficiency for linux kernel fuzzing. Journal of Physics: Conference Series, 1176:022013.

Liang, J., Wang, M., Zhou, C., Wu, Z., Jiang, Y., Liu, J., Liu, Z., and Sun, J. (2022). Pata: Fuzzing with path aware taint analysis. In 2022 IEEE Symposium on Security and Privacy (SP), pages 1–17.

Lou, B., & Song, J. (2020). A Study on Using Code Coverage Information Extracted from Binary to Guide Fuzzing.International Journal of Computer Science and Security (IJCSS), Volume (14): Issue (5): 2020.

Luk, C.K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. (2005). Pin: Building customized program analysis tools with dynamic instrumentation. In Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '05, pages 190–200, New York, NY, USA.

Manes, V. J. M., Han, H., Han, C., Cha, S. K., Egele, M., Schwartz, E. J., and Woo, M. (2019). The art, science, and engineering of fuzzing: A survey. IEEE Transactions on Software Engineering.

Miller, B. P., Fredriksen, L., and So, B. (1990). An empirical study of the reliability of unix utilities. Commun. ACM, 33(12):32–44.

Mouzarani, M., Sadeghiyan, B., and Zolfaghari, M. (2015). A smart fuzzing method for detecting heap-based buffer overflow in executable codes. In 21st IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2015, Zhangjiajie, China, November 2015, pages 42–49.

Oehlert, P. (2005). Violating assumptions with fuzzing. IEEE Security and Privacy, 3(2):58–62.

Ognawala, S., Hutzelmann, T., Psallida, E., and Pretschner, A. (2017). Improving function coverage with munch: A hybrid fuzzing and directed symbolic execution approach. CoRR, abs/1711.09362.

Petsios, T., Zhao, J., Keromytis, A. D., and Jana, S. (2017). Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. CoRR, abs/1708.08437.

Salem, H. A. and Song, J. (2021). Using grammar extracted from sample inputs to generate effective fuzzing files.

Sanitizer coverage. (2017). https://clang.llvm.org/docs/SanitizerCoverage.html.

Stephens, N., Grosen, J., Salls, C., Dutcher, A., Wang, R., Corbetta, J., Shoshitaishvili, Y., Kruegel, C., and Vigna, G. (2016). Driller: Augmenting fuzzing through selective symbolic execution. In 23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016.

Suh, G. E., Lee, J. W., Zhang, D., and Devadas, S. (2004). Secure program execution via dynamic information flow tracking. SIGARCH Comput. Archit. News, 32(5):85–96.

Sutton, M., Greene, A., and Amini, P. (2007). Fuzzing: Brute Force Vulnerability Discovery. Addison-Wesley Professional.

Syzkaller. (2017). syzkaller - linuxsyscall fuzzer. https://github.com/google/syzkaller.

Trail of Bits. (2016). Shin grr: Make fuzzing fast again. https://blog.trailofbits.com/2016/11/02/shin-grr-make-fuzzing-fast-again/.

Wang, S. and Wu, D. (2017). In-memory fuzzing for binary code similarity analysis. In 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE), pages 319–330.

Yun, I., Lee, S., Xu, M., Jang, Y., and Kim, T. (2018). Qsym: A practical concolic execution engine tailored for hybrid fuzzing. In Proceedings of the 27th USENIX Conference on Security Symposium, SEC'18, page 745–761, USA. USENIX Association.

Zalewski, M. (2013) American fuzzy lop. http://lcamtuf.coredump.cx/afl/.

Zalewski, M. (2015) New in afl: persistent mode. https://lcamtuf.blogspot.com/2015/06/new-in-afl-persistent-mode.html.