A Study on Using Code Coverage Information Extracted from Binary to Guide Fuzzing

Baoying Lou *lou1407@vandals.uidaho.edu*
Computer Science Department
University of Idaho
Moscow, ID, 83844, USA

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

Abstract

Code coverage is commonly used in software testing because it tells which portion of code has been tested or not. Fuzzing is one of the most popular and powerful solutions to find software vulnerabilities. And code coverage information is used in several fuzzing techniques to guide the testing. Coverage-guided fuzzer is efficient and effective by tracking and utilizing code coverage feedback. In practice, when the source code of a target application is not provided, we have to focus on the binary files and fuzz the executable files. This paper briefly introduces fuzzing techniques and the common code coverage measurement criteria. Then the paper give a comprehensive review and summary of the ways to gather coverage information, including source code instrumentation, dynamic instrumentation, static instrumentation, emulation, debugger, and hardware feature. Their advantages and disadvantages are discussed. Few studies have been conducted on the techniques that fuzzers extract code coverage information from binary files and use it to guide fuzzers in next step. Therefore this paper also provides a summary of how fuzzers utilize code coverage feedback information and what are the strengths and limitations of each of them.

Keywords: Code Coverage, Fuzzing, Software Testing, Binary Analysis, Test Case Generation.

1. INTRODUCTION

In software development life cycle, testing is a crucial step to ensure the security and quality of software. With the increasing size and complexity of software, cyber threats on software security have been prevailing and have increased exponentially [33], and adequate software testing has become increasingly important [34]. Code coverage is commonly used in software testing because it gives a picture of which portion of the code has been tested. Code coverage keeps track of the portion of code or paths that have been covered and the part which has not been reached. Therefore code coverage is commonly used in the software testing to guide the testing toward the portion that has not been tested. In addition, it is often used as a yardstick to measure the efficiency and adequacy of a testing [1]. It can help testers to understand how much code is tested in a quantitative measurement.

Code coverage can be used for software testing, unit testing, integration testing, and usage analysis. Therefore a code coverage tool is usually integrated into various tools, such as IDE (Eclipse, NetBeans…), or Build tool (Maven, Gradle, Ant), or CI (Jeakins, TeamCity, Visual Studio Team Services) [2]. Code coverage can tell us how much of the code is tested, however, good code coverage does not mean good testing. For instance, even if the code coverage shows a testing has covered 100% of the code, it may not be able to reveal some or all of the existing vulnerabilities. Therefore a good coverage in testing does not mean it is a complete testing. And

passing the testing with 100% code coverage does not mean the software under testing is perfect without any vulnerability.

Code coverage can also be used for fuzzing. Fuzzing is one of the software testing methods which is widely used by developers and testers. In traditional fuzzing, random inputs are fed into the program, intend to trigger abnormal behavior of the program, such as crash or unexpected exception. With fuzzing, the whole testing process can be automated and it does not need much time to set up the environment. However the drawback of fuzzing is that it can reveal the easy-to-reach bugs, not the bugs located at deeper level in the software. To make fuzzers smarter, researchers have integrated various techniques in fuzzers to guide the fuzz testing, such as symbolic execution, concolic execution, grammar, taint analysis, code coverage, machine learning, and so on [3].

In this paper, we give an overview of how fuzzers gather code coverage information and how code coverage feedback is used to improve the efficiency and effectiveness of coverage-guided fuzzing tools. Besides, we analyze and compare a few popular and wide-used fuzzers and different techniques they use to gather and use code coverage feedback.

In the remainder of this paper, Section 2 introduces fuzzing techniques and code coverage criteria. Section 3 discusses the ways of collecting coverage information at the software level and hardware level. A detailed study on how code coverage feedback is used in fuzzing tools is provided in Section 4. The usages of code coverage information in different fuzzing tools are summarized at the end of that section. And section 5 concludes the paper.

2. BACKGROUND

This section introduces fuzzing techniques, code coverage, and coverage measurement criteria.

1. Fuzzing Techniques

Fuzzing was developed by Miller at the University of Wisconsin in 1989 [4]. It was a black box testing strategy that does not require access to the source code. Fuzzers feed a program with random inputs and monitor the abnormal behavior of the program, including crashes, memory leaks, unhandled exception, etc. Most inputs are invalid and rejected by a fuzzer immediately. One way to solve this problem is to introduce small changes to existing well-formed inputs so that the new input might still be valid [32].

Fuzzing is much faster than manual source code review, and it can run 24 hours / 7 days [5]. However, it usually detects simple crashes, not deep ones. Nowadays, fuzzing is commonly used for security testing. For example, Microsoft security development lifecycle (SDL) process is a software security assurance process to help developer build high assurance software. SDL utilizes black-box fuzzing in its verification phase [6].

According to the test case generation techniques a fuzzer uses, existing fuzzers can be categorized into three groups: blind fuzzers, coverage-guided fuzzers, hybrid fuzzers [7].

1.1. Blind Fuzzer

Blind fuzzers are also known as dumb fuzzers. A blind fuzzer mutates the existing input blindly. It doesn't use any feedback from previous tests. It generates a lot of random and invalid inputs that are rejected by the program quickly. For example, PEACH, RADAMSA, ZZUF are blind fuzzers.

PEACH has both generation and mutation capabilities. It works by creating PeachPit files. PeachPit files are XML files containing the complete information about the data structure and type information.

1.2. Coverage-guided Fuzzer
A fuzzer that uses code coverage feedback information is categorized as a coverage-guided fuzzer. Most inputs generated by blind fuzzer are invalid and would be rejected by the program directly. A coverage-guided fuzzer can mutate one valid input file iteratively to increase the code coverage. More information about coverage-guided fuzzers provided in section 4. AFL [8], HONGGFUZZ [9], VUZZER [10] are widely used coverage-guided fuzzers.

1.3. Hybrid Fuzzer
Hybrid fuzzer use symbolic execution or taint analysis to uncover code edges that coverage-guided fuzzer cannot reach [7]. A hybrid fuzzer calculates and extracts the correct input necessary for new code coverage, such as DRILLER [11], T-FUZZ [12]. The main idea of symbolic execution is to use symbolic values instead of concrete values as input values and to use symbolic expressions to represent the values of program variables [38]. In software testing, symbolic execution is typically used to explore as many program paths as possible. In dynamic taint analysis, all the data directly from or derived from untrusted sources, which could be fuzzed inputs, are labeled as tainted [39]. These tainted data and the data propagated from the tainted data are tracked during program execution, and insecure use of the data is detected.

A binary file is computer readable, but not human-readable. In computer science, executable files are usually stored in binary files. When the source file is not available, we have to do binary analysis. Many fuzzers provide their own compilers, such as AFL has afl-gcc, and HONGFUZZ has hfuzz. These compilers insert instrumentation code during compilation to track code coverage. This strategy is known as source code instrumentation. For binary files, fuzzers have to utilize some different strategies rather than source code instrumentation to obtain coverage information if they want to use the code coverage feedback, such as static binary instrumentation or dynamic instrumentation. More details are given in section 3.1.
A comparison of different types of fuzzers is shown in Table 1.

CATEGORY	PROS	CONS	FUZZER
Blind fuzzer	runs fast	produces low quality input	PEACH, RADAMSA, ZZZUF
Coverage-guided fuzzer	good code coverage	can not find very deep bugs	AFL, HONGFUZZ, VUZZER, ANGORA
Hybrid fuzzer	better code coverage	runs slow, not scalable, can't solve many types of constraints efficiently	DRILLER, T-FUZZ

TABLE 1: Comparison of Different Types of Fuzzers.

2. Code Coverage and Coverage Measurement Criteria
Code coverage provides a quantitative measure of how thoroughly a test is. Code coverage could also help figure out the parts of a program that are not reached by a set of test cases, and it can help create extra test cases using the coverage information to increase coverage.

Coverage measures the amount of testing done based on a certain criterion [13]. A coverage criterion is a rule or collection of rules that impose test requirements on a test set. Kaner has summarized 101 types of testing coverage [13]. Usually, these metrics are calculated by the number of items triggered and tested divide by the total number of items existing in the program. The major coverage measures are listed below.

2.1. Statement Coverage/Line Coverage
Statement coverage is also called line coverage. It is the most basic and simplest coverage criteria in the white box testing. It is used to calculate and measure the number of statements in the source code which are executed at least once given the input. Statement coverage is the

most used kind of coverage criterion in the industry [14]. It is often used by developers to evaluate the quality of programs.

2.2. Block Coverage

Block coverage or basic block coverage describes a block of code, defined as not having any branch point within, is executed or not. In other words, a basic block always executes as one atomic unit without any jumps and jump targets. Several lines of source code could be in the same basic block. It makes more sense to keep track of basic blocks rather than individual lines for efficiency reasons at execution time [15]. For example, if there is one line in an if block, but there are nine lines of code in the else block, it will be 10% line coverage if the if branch is executed, while 50% block coverage if the else branch is executed. Obviously, block coverage is more desirable than line coverage.

2.3. Branch Coverage

Branch coverage also is known as decision coverage. It equals the number of executed statement blocks and decisions divided by the total number of statements and decisions. Each decision counts twice, one for the true case and one for the false case. Branch coverage is widely used because of its ease of implementation and its low overhead on the execution of the program under test.

Examples of branch or decision statements are switch, do-while, and if-else statements.

There are two main weakness of branch coverage. First weakness of branch coverage is that only the outcome of boolean expressions are considered. For example, in the tiny program, the outcome of a is executed, but the outcome of `!a` is not considered. The line coverage is 100% when a is true, which branch coverage is 50%.

```
if (a) {
  print(a);
}
```

Another limitation of branch coverage is that it is not applicable if a program has no decision statement, such as if-else, do-while loop.

2.4. Path Coverage

Path coverage is the ratio of all paths through the control flow graph covered and the number of total paths. In practice, path coverage is impossible. If there are *n* decisions to make in the control flow graph, it could result in 2^n paths. The loop (while loop, for loop) will make things worse. Furthermore, some code paths could not be executed anyway since there may be no input that can trigger that path.

2.5. Function/Method Coverage

Method coverage measures how many of the functions/methods in the program have been called during execution. It is the easiest criteria to measure code coverage.

3. COLLECTING CODE COVERAGE INFORMATION

Code coverage can be measured by instrumenting the source code or binary code, as well as using hardware features. Methods to measure code coverage are summarized in Table 1 and detailed here [7]:

3.1 Source Code Instrumentation

Source code instrumentation automatically adds specific code to the source files under analysis. After compilation, execution of the code produces dump data for runtime analysis or component testing. Source code instrumentation is widely used by coverage-guided fuzzers, such as AFL [8], HONGFUZZ [9], T-FUZZ [11], VUZZER [10].

Source code instrumentation is the most powerful, flexible and accurate way to calculate code coverage. It can be done for function coverage, basic blocks coverage, statement coverage, and branch coverage, since it is implemented at the source-code level.

Source code instrumentation will be inefficient if the target has a large amount of code. It will be impossible if the source code is not available. It also needs a specific compiler to compile an instrumented code, So it is not portable across different programming languages [16].

3.2 Dynamic Binary Instrumentation

Dynamic binary instrumentation (DBI) is a method of binary analysis at runtime by injection of instrumentation code to a binary file to collect information without requiring access to its source code or modifications to the runtime. DBI is usually used to conduct program analysis or architectural studies, such as code coverage, call-graph generation, memory leak detection, and fault injection [17]. DBI is especially necessary for application analysis on Windows system, because most windows programs are not open source.

There are two approaches to dynamic binary instrumentation. The first one is dynamic binary translation (DBT), which copies the target binary in segments and instruments the copies during runtime. The second one is dynamic probe injection (DPI) [16]. DBI process the target binary in memory to insert probes that lead execution to the instrumentation. Comparing to DBT, DPI doesn't have copying overhead. It instruments the binary directly. But it could cause performance overhead from preserving target program behavior while inserting probes, as well as costs from context-switching and redirecting of execution by the probes [16].

DBI has many advantages. It is non-bypassable instrumentation, complete and easy to use [18]. We could use DBI platforms such as Pin, DynamoRIO, and Valgrind to develop dynamic binary instrumentation tools so that we don't have to build it from scratch.

Even though dynamic binary instrumentation is used widely, it is impractical to instrument a whole target binary because of overhead. Usually, a specific code area or a specific function is chosen to be instrumented. Library functions that we do not care about, such as printf(), should not be considered [19].

3.3 Static Binary Instrumentation

Static binary instrumentation (SBI) is also as known as binary rewriting, which refers to rewriting the compiled binary statically [20]. Compared with dynamic binary instrumentation, SBI is more efficient than DBI since SBI doesn't introduce as much runtime performance overhead as DBI and all the instrumentation code is added prior to execution [21].

One challenge of static binary instrumentation is to differentiate code and data. Current approaches to static rewriting can only provide limited coverage and reduced accuracy [22].

3.4 Emulation of Binary

QEMU works by compiling the target object code into the host object code and that the host computer is typically faster than the target one, virtualization is actually a plus over direct execution on the target [23].

3.5 Writing Own Debugger and Set Breakpoints On Every Basic Block

It runs slow, but useful in some situations [5]. First, we need to write a specific debugger. Set breakpoints to every single basic block in the program via the debugger. These breakpoints are removed as they are hit. In this way, the debugger can gather the coverage information about the binary [24]. This method could be very slow because the debugger processes many switches which takes a lot of time [5].

3.6 Use of Hardware Features

Another method is to collect code coverage from physical hardware. For example, Intel Processor Trace (Intel PT) is a new feature built into some Intel processors that let us collect execution and branch trace information from CPU cores. Intel PT is implemented entirely in hardware with very little overhead, and no instrumentation in software is needed [25].

Except for source code instrumentation, all other above methods or combinations could be used for binary code coverage. Some fuzzers use dynamic binary instrumentation to obtain code coverage information when performing fuzzing with binary files. There are a few fuzzers that use other strategies. For example, AFL uses QEMU for dynamic binary instrumentation and GCC and Clang compilers for static source instrumentation. VUzzer uses PIN for dynamic binary instrumentation. kAFL uses QEMU and Intel processor trace to fuzz OS kernel [26].

A summary of different methods to collect code coverage information is shown in Table 2.

METHOD	PROS	CONS
Source code instrumentation	Powerful, flexible, accurate	Unportable, source code required
Dynamic binary instrumentation	Non-bypassable, completeness, ease of use	difficult to implement, overhead at run-time
Static binary instrumentation	Low overhead at run-time	Break the application if not differentiate data and code properly, low accuracy
Own Debugger or set breakpoints	Language-independent, non-intrusive	slow because of debugger process switches
Hardware features	Low-overhead, non-intrusive, language-independent, most accurate	Hardware dependent

TABLE 2: Comparison of different methods of collecting code coverage information.

4. THE USE OF COVERAGE FEEDBACK IN FUZZING

Fuzzing is one of the most powerful techniques to find vulnerabilities in software. But fuzzers can only find shallow bugs, not deeper ones. There are many attempts to deal with this problem. Such as corpus distillation, symbolic execution, etc. In this section, we mainly focus on the fuzzers that use code-coverage feedback, such as AFL, VUzzer, honggfuzz.

In general, a coverage-guided fuzzer's algorithm contains these steps [27]:

Step 1: Load initial test cases into the poll or queue. initial test cases are created by users.
Step 2: Take the next seed test cases from the queue.
Step 3: Repeatedly mutate this test file to generate a batch of new test cases.
Step 4: Apply mutations to the target application.
Step 5: Monitor the behavior of the target application and track the code coverage.
Step 6: Filtrate good mutants that trigger new code area to the queue.
Step 7: Go to step 2 literately.

Different coverage-guided fuzzers use different code coverage strategies to fuzz binary files. AFL uses static instrumentation provided by compiler and dynamic instrumentation with QEMU to track edge coverage. HongFuzzer uses SanitizerCoverage instrumentation method to track block coverage [9]. VUzzer uses PIN to track block coverage [10].

4.1 American Fuzzy Lop (AFL)

AFL is a popular fuzzer used by many developers and researchers. To keep track of the coverage information, AFL maintains a bitmap. Each byte in the bitmap represents the number of times a branch has been taken. AFL assigns a random two-byte ID to each branch during instrumentation. When executing a branch, AFL does an XOR of the last branch ID with the current branch ID. Then it calculates the hash value of the XOR'd value by using a hash function. Next it finds the corresponding entry that represents these two branches combination, and increases the byte value by one in the bitmap [28]. Since hash function is used in this process, it could cause hash collision, which could make AFL mistakes a new path for an existing path in the bitmap.

Hash collision could prevent the fuzzer to find more paths and give a lower code coverage feedback information. For example, if there is a good test case which can trigger a new code path, but AFL categorizes it as another path in the map, then this test case will be discarded and will not be put back into the queue for the next fuzzing round. Many researchers have been working on solving this problem and improve AFL, such as CollAFL [27], which is a coverage sensitive fuzzing solution. It could reduce hash collision significantly for source code fuzzing. Hash collision increases to perform fuzzing on binary files than source code.

After one round, AFL will pull one test case from the queue and mutate it according to some strategies [29], such as walking bit flips, walking byte flips, simple arithmetic, known integers, stacked tweaks, and test case splicing.

4.2 ANGORA

ANGORA is a mutation-based fuzzer that uses some key techniques to produce high quality inputs, such as scalable byte-level taint tracking, context-sensitive branch count, search algorithm based on gradient descent, shape and type inference, and input length exploration [30].

ANGORA utilizes a very similar scheme with AFL with a little difference. When AFL performs XOR calculation, it takes the last branch and current branch into consideration. ANGORA considers the current branch and a hash of the call stack to do XOR, so that ANGORA can tell the same coverage in different contexts [31]. In this way, the hash collision problem could be solved.

4.3 VUzzer

VUzzer is an application-aware evolutionary fuzzer. It doesn't require any prior knowledge of the application or input format [10]. It uses control flow and data flow features based on static and dynamic analysis to maximize coverage and explore deeper paths, so it can generate qualified inputs and discover bugs deep in the program [10].

VUzzer is very different from AFL and ANGORA. Unlike AFL using a bitmap, VUzzer assigns scores to each basic block according to the depth they are within a function. Deeper blocks have higher scores and shallow blocks have lower scores. Finally, it sums all the scores of all basic blocks in a path to get a fitness score. VUzzer utilizes an evolutionary algorithm to produce new mutations. If one test case has a high fitness score, it could produce more offsprings.

As shown in Table 3, a comparison of how code coverage feedback is used in each fuzzing tool is provided.

FUZZER	STRATEGY	PROS	CONS
AFL	Bitmap	Fast	Hash collsion
ANGORA	Bitmap and call stack	Less hash collision	More memory usage
VUzzer	Weight-aware fitness strategy	No hash collision, good performance	Overhead in computing the fitness score

TABLE 3: Comparison of How Fuzzers Use Code Coverage Feedback.

5. CONCLUSION

This paper summarizes common code coverage measurement criteria, such as statement coverage, block coverage, branch coverage, path coverage, and function coverage. We also studied the methods to gather code coverage information and listed their pros and cons. Each method has its pros and cons. All methods can be used for binary files except source code instrumentation. In practice, most fuzzers utilize dynamic binary instrumentation to track code coverage when fuzzing binary file, but dynamic binary instrumentation has significant run-time overhead. Source code instrumentation has a lower overhead than dynamic binary

instrumentation. If we have access to source code, source code instrumentation is a good choice. Using hardware features to obtain code coverage seems a perfect method. It is non-intrusive, language-independent, most accurate, and it has low overhead, however it requires hardware support.

We analyzed how fuzzers utilize code coverage information in this paper. The core algorithm of AFL uses a bitmap to trace block transition when encountered. Since a hash function is used, hash collision can happen if the size of bitmap is not big enough. The code coverage may be not accurate enough due to hash collision. Accurate code coverage information could help fuzzers explore unique paths and find more vulnerabilities. If a bigger bitmap is used, it lowers the hash collision rate. However, maintaining a bigger bitmap reduces the fuzzer's performance at the same time. Sometimes fuzzers need to trade off the code coverage accuracy with better performance. ANGORA uses a bigger bitmap and a call stack to solve the hash collision problem. VUzzer uses a different strategy than AFL and ANGORA. It calculates the fitness score of inputs. If one input exercises deep code block, it has a higher fitness score, and it can produce more offspring in the next generation round.

Fuzzing is an efficient way to detecting vulnerabilities, but it also has disadvantages, and there are still a lot of challenges need to be solved. For example, fuzzing is not exhaustive. It could happen that an input covers a piece of code without revealing bugs in it, because the fuzzer treat covered code equally. Some new techniques could be used to solve this problem and improve fuzzing, such as machine learning [35] and coverage accounting [36]. TortoiseFuzz uses coverage accounting which is novel approach to evaluate coverage by security impacts. It could combine the evaluation with coverage information for input prioritization [36]. Code coverage fuzzing also could be time consuming. Code coverage tracing is a dominant source of overhead. Untracer uses coverage-guided tracing technique to eliminate needless tracing [37].

6. REFERENCES

[1] P. S. Kochhar, F. Thung, D. Lo. "Code Coverage and Test Suite Effectiveness: Empirical Study with Real Bugs in Large Systems," In IEEE Symposium on Security and Privacy, 2015.

[2] G. J. Myers. "The Art of Software Testing," Second Edition. 2004.

[3] H. Al Salem and J. Song, "A Review on Grammar-Based Fuzzing Techniques", International Journal of Computer Science & Security (IJCSS), Volume (13), Issue (3), 2019.

[4] Wikipedia, Fuzzing, https://en.wikipedia.org/wiki/Fuzzing, Dec. 3, 2020 [Dec 4, 2020].

[5] R. Freingruber, "The art of fuzzing", SEC Consult, 2017.

[6] Microsoft Security Engineering, https://www.microsoft.com/en-us/securityengineering/sdl [Dec 4, 2020].

[7] E. Guler, C. Aschermann, Ali Abbasi, and Thorsten Holz. "AntiFuzz: Impeding Fuzzing Audits of Binary Executables". In USENIX Security Symposium, 2019.

[8] M. Zalewski, "American fuzzy lop", https://lcamtuf.coredump.cx/afl/ [Dec 4, 2020].

[9] Honggfuzz, https://github.com/google/honggfuzz, Dec, 2, 2020 [Dec 4, 2020].

[10] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida and H. Bos, "VUzzer: Application-aware Evolutionary Fuzzing" NDSS, Feb. 2017.

[11] N. Stephens and John Grosen and C. Salls and Andrew Dutcher and Ruoyu Wang and Jacopo Corbetta and Yan Shoshitaishvili and C. Krugel and G. Vigna. "Driller: Augmenting Fuzzing Through Selective Symbolic Execution", NDSS, 2016.

[12] H. Peng and Y. Shoshitaishvili and M. Payer. "T-Fuzz: Fuzzing by Program Transformation", 2018 IEEE Symposium on Security and Privacy (SP), pages 697-710.

[13] C. Kaner, "Software negligence and testing coverage". Software QA Quarterly, vol.2, #2, pp.18, 1995.

[14] Testing Brain, "Statement Coverage in software testing". https://www.testingbrain.com/whitebox/statement-coverage.html [Dec 4, 2020].

[15] Emma, "EMMA: Frequently asked questions". http://emma.sourceforge.net/faq.html, Jan, 1, 2006 [Dec 4, 2020].

[16] V. Zhao, "Evaluation of Dynamic Binary Instrumentation Approaches: Dynamic Binary Translation vs. Dynamic Probe Injection " Honors Thesis Collection. 575. 2018.

[17] M. Young, "The Technical Writer's Handbook". Mill Valley, CA: University Science, 1989.

[18] D. C. D'Elia, "SoK: Using Dynamic Binary Instrumentation for Security". In ACM Asia Conference on Computer and Communications Security (AsiaCCS '19), July 9–12, 2019, Auckland, New Zealand. ACM, New York, NY, USA, 2019.

[19] J. Salwan, "Dynamic Binary Analysis and Instrumentation Covering a function using a DSE approach", 2015.

[20] D. C. D'Elia, E. Coppa, S. Nicchi, F. Palmaro, and L. Cavallaro, "SoK: Using Dynamic Binary Instrumentation for Security (And How You May Get Caught Red Handed). In ACM Asia Conference on Computer and Communications Security (AsiaCCS '19), July 9–12, 2019.

[21] M. A. Laurenzano, M. M. Tikir, L. Carrington and A. Snavely, "PEBIL: Efficient static binary instrumentation for Linux," 2010 IEEE International Symposium on Performance Analysis of Systems & Software (ISPASS), White Plains, NY, 2010.

[22] M. Smithson, K. ElWazeer, K. Anand, A. Kotha, R. Barua, "Static Binary Rewriting without Supplemental Information". WCRE, Koblenz, Germany, 2013.

[23] "QEMU documentation". https://www.qemu.org/docs/master/ [Dec 4, 2020].

[24] "gamozolabs / mesos". https://github.com/gamozolabs/mesos [Dec 4, 2020].

[25] Intel Developer Zone, "Online Guide for the Intel Joule Module". https://software.intel.com/en-us/node/721535 [Dec 4, 2020].

[26] RAPITA Systems, "Code coverage without instrumentation". https://www.rapitasystems.com/blog/code-coverage-without-instrumentation [Dec 4, 2020].

[27] S. Gan, C. Zhang, X. Qin, X. Tu, K. Tu, K. Li, Z. Pei, Z. Chen, "CollAFL: Path Sensitive Fuzzing". IEEE Symposium on Security and Privacy (SP), 2018.

[28] A. Henderson, H. Yin, G. Jin, H. Han, Deng H. "VDF: Targeted Evolutionary Fuzz Testing of Virtual Devices". In: Dacier M., Bailey M., Polychronakis M., Antonakakis M. (eds) Research in Attacks, Intrusions, and Defenses. RAID 2017. Lecture Notes in Computer Science, vol 10453. Springer, Cham, 2017.

[29] lcamtuf's blog, "Binary fuzzing strategies: what works, what doesn't". https://lcamtuf.blogspot.com/2014/08/binary-fuzzing-strategies-what-works.html [Dec 4, 2020].

[30] P. Chen and H. Chen. "Angora: Efficient Fuzzing by Principled Search" In IEEE Symposium on Security and Privacy (S&P), 2018.

[31] S. Schumilo, C. Aschermann, R. Gawlik, S. Schinzel, T. Holz, "kAFL: Hardware-Assisted Feedback Fuzzing for OS Kernels", 26th USENIX Security Symposium, 2017.

[32] H. A. Salem, J. Song, “A Review on Grammar-Based Fuzzing Techniques”, International Journal of Computer Science & Security (IJCSS), 2019.

[33] A. Yeboah-Ofori, “Software Reliability and Quality Assurance Challenges in Cyber Physical Systems Security ”, International Journal of Computer Science and Security (IJCSS), Volume (14) : Issue (3) : 2020.

[34] Mohd. Ehmer Khan, “Different Software Testing Levels for Detecting Errors ”, International Journal of Software Engineering (IJSE), Volume (2) : Issue (4) : 2011.

[35] Jun Li, Bodong Zhao, and Chao Zhang. 2018. “Fuzzing: a Survey”. Cybersecurity, 2018

[36] Y. Wang, X. Jia, Y. Liu, K. Zeng, T. Bao, D. Wu, and P. Su. “Not all coverage measurements are equal: Fuzzing by coverage accounting for input prioritization”. In NDSS’ 20, 2020.

[37] S. Nagy and M. Hicks, "Full-Speed Fuzzing: Reducing Fuzzing Overhead through Coverage-Guided Tracing," 2019 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 2019.

[38] Salahaldeen Duraibi*, Abdullah Alashjaee*, Jia Song, “A Survey of Symbolic Execution Tools”, International Journal of Computer Science and Security (IJCSS) Volume-13 Issue-6, 2019.

[39] Abdullah Alashjaee*, Salahaldeen Duraibi*, Jia Song, “Dynamic Taint Analysis Tools: A Review”, International Journal of Computer Science and Security (IJCSS) Volume-13 Issue-6, 2019.