Lab Malware Analysis Report

Coverage-Guided Fuzzer in Python for Source Code Programs

Winter Semester 2018/19: Block Course

Author: Kainaat Singh(3205599)

Advisor: Daniel Baier

17 April 2019

Contents

List of Algorithms

List of Figures

List of Tables

1 Introduction

Security Vulnerabilities within a computer system are weaknesses that symbolize great risks by leaving the system open to attack. Network personnel and users are always trying to defeat the dangers posed by these vulnerabilities [4]. There are many different approaches to finding vulnerabilities, but broadly they are divided into white, black and gray-box testing. Each of the methods comes with its own set of advantages and disadvantages.

White box testing [19] uses all possible resources to get an internal outlook of the system in question, including the source code. Under this, analysis of the source code is performed either statically or dynamically by using automated tools. It is possible to cover all possible paths to check for vulnerabilities as the source code is accessible. But, in practice it has a high false positive rate and due to the complexity of real-world applications, it can be a time-consuming process and, the source code is generally not available.

Black box testing [19] provides the tester with the knowledge of only the inputs and their corresponding outputs. The tester is entirely oblivious to the inner workings of the system under test. Black box testing can be carried out statically or dynamically by automated Fuzzing, in which random inputs are thrown at the target, and the results are monitored. This form of testing is simple to carry out, applicable in every situation and is reproducible to test another system. Despite this, the biggest hurdle is to determine when to stop testing and how successful the testing has been.

Gray box testing [19] involves the process of black box testing along with the added knowledge received through reverse code engineering(RCE). Due to the inaccessible source code, the static analysis of machine instructions in the binary can help discover the vulnerabilities, but with a lot more effort involved. Dynamic analysis tools can also be used to analyze the binaries, and further fuzzing can use this data to find vulnerabilities. Gray box testing increases the coverage that achieved through black box techniques but, the complexity of the testing also increases with the inclusion of RCE.

Apart from the above, Symbolic Execution [13] is another popular method to finding vulnerabilities. This method considers program inputs as symbols and associates every execution path to a set of constraints. It makes use of constraint solvers to determine which symbol caused the program to crash. This method can lead to uncovering of all execution paths of a program which has been proven to be effective for small programs. Though, as the size of the program increases, the path explosion problem occurs.

Among the different testing methodologies stated, Fuzzing has the highest ratio of automation to manual labor [19]. Hence, it is a highly popular method to find vulnerabilities.

1.1 Background Knowledge

In this section, we have tried to provide a broader perspective on fuzzing by defining essential terms, working process and the challenges faced while fuzzing.

Fuzzing
Fuzzing [14] is defined as an art of finding vulnerabilities by running the System Under Test with "fuzz inputs". According to [19] fuzz input is an input that can provoke an unexpected behavior from the system under test.

Fuzz testing [14] is a quality assurance testing that uses the concept of fuzzing to test a system for violation of a security policy when it processes a malicious input. Fuzzer is the security tool used by testers that performs fuzz testing on the system to find bugs.

Working Algorithm of Fuzzing
There are various fuzzing approaches, and there is no single correct approach. It can rely on the type of target, the proficiency of the tester and the type of data we fuzz. However, in [14] a study on various fuzzers was conducted, and a standard algorithm was presented that sums up some basic phases of fuzzing that might apply.

Algorithm 1: Fuzz Testing Algorithm

$Input : (C, t_{limit})$;
$Output$: B //a finite set of bugs;
$B \leftarrow \emptyset$;
$C \leftarrow$ Preprocess(C);
while $t_{elapsed} < t_{limit}$ **do**
  $conf \leftarrow$ Schedule($C, t_{elapsed}, t_{limit}$);
  $tcs \leftarrow$ InputGen($conf$);
  $B', execinfos \leftarrow$ InputEval($conf, tcs, Obug$);
  $C \leftarrow$ ConfUpdate($C, conf, execinfos$);
  $B \leftarrow B \cup B'$;
end while
return B;

Note: Reprinted from Fuzzing: Art, Science, and Engineering by Valentin J.M. Manes, retrieved from https://arxiv.org/abs/1812.00140 Copyright 2019 by arXiv:1812.00140v3 [cs.CR]

Table 1 depicts a traditional fuzz testing algorithm. The algorithm accepts a set of valid fuzz inputs C and the timeout limit of each run tlimit as input and reports an output of the found bugs [14]. The quality of the fuzz inputs provided is in direct correlation to the test case effects [13]. On the one hand, they need to be following the input format of the system under test, and on the other hand, they need to be broken enough so that they can extract unexpected behavior from the system.

The working model consists of two parts. The first part is Preprocess in which the fuzzer performs modification on the provided fuzz inputs. Some fuzzers use instrumentation on the system under test which helps them to decide on which seed value of the fuzz input is beneficial for testing and which seed value can be discarded. This problem is widely regarded as seed selection problem [14].

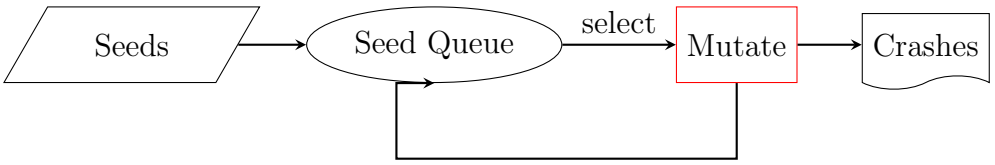
The second part of the algorithm has five functions inside a loop. Each iteration of this loop is called a fuzz iteration. The first function Schedule accepts the queue containing the fuzz inputs C received, timeout limit tlimit and calculates the current time telapsed. It selects a fuzz input C to be used for this loop iteration. The second function InputGen is the most influential function of the fuzzer. It accepts a fuzz input C and provides a set of test cases as output. The third function InputEval which accepts a set of test cases as input and executes the system under test with each of the test cases conf and monitors which test case violates the security policy. The fourth function ConfUpdate accepts the set of fuzz inputs C, current fuzz input conf and the information of each execution. It can update the fuzz inputs C to make the fuzzing more efficient by discarding some fuzz inputs C based on information of each fuzz run. The fifth function Continue checks whether the next iteration should take place or not. This function is essential to stop when no new paths can be found [14].

Types of Fuzzers

There are different ways in which fuzzers can be classified. [13] The first type of classification is dependent on the type of input generation method deployed. All fuzzers under this classification fall under mutation based or generation based. Mutation-based fuzzers apply mutations to existing fuzz inputs to create test cases. Generation based fuzzers create test cases from scratch by modeling the target protocol.

The second type of classification is with respect to source code availability and amount of program analysis involved [13]. The fuzzers can be classified as white box, black box or grey box fuzzers. White box fuzzers have access to source code and hence have more control over the information retrieved through static analysis. Black box fuzzers have no source code and get results

Figure 1: Simple Flow Diagram of a Fuzzer



by observing the behaviour of inputs and the corresponding output. Grey box fuzzers also do not have access to the source code but, they get the information on the internal workings of the system through analysis.

The third type of classification is dependent on whether there is execution feedback that helps to evolve test case generation [13]. Fuzzers can be classified under Dumb or Smart. A dumb fuzzer has no existing information about the fuzz input and mutates it randomly. In a smart fuzzer, the difference lies in the way the fuzz input is mutated. The fuzzer must understand the input format, and based on it; it makes an informed decision on the method of mutating the individual elements.

Challenges Faced in Fuzzing

In recent years, fuzzing has become extremely popular with great fuzzing solutions like AFL. It has been identified as a fast technique that can discover real bugs, but it has some flaws that still need to be addressed.

The first challenge is generating effective test cases [13]. This issue stems from the question that how should the input be mutated so that more complex program code is covered as the randomly generated inputs can fail complex checks and hence, fail to cover the code. For mutation based fuzzers, this is a significant challenge to resolve as to how to mutate the input and where to mutate the input so that we can have a set of effective test cases that can help make the fuzzing more efficient. There is a need to avoid wastage of testing resources by improving the effectiveness of test cases.

The second prominent challenge faced is low code coverage [13]. It has been proven in previous works that better coverage of code consequently leads to discovering more number of bugs. This is because higher the coverage means more code was tested during execution. Various Coverage-based fuzzers try to overcome this challenge. In the next section, I shall shed more light on this issue.

The third challenge faced is passing complex sanity checks [13]. These checks are placed to avoid malicious inputs that could harm the system and the program. Many fuzzers blindly generate random inputs which do not pass these checks.

Coverage-Guided Fuzzing
Many successful fuzzers use the coverage-guided method and have proved it to be quite an effective method. To make a fuzzer more efficient, as many as possible program states should be covered. There is no single standard to test whether all program states have been reached due to uncertain program behaviors. Code coverage acts as an approximate solution to this problem. In this context, increasing coverage leads to an increased number of program states visited [13]. Hence, the motivation to develop a coverage-guided fuzzer.

The coverage guided fuzzing works as follows[15]: Initialize the seed queue S. Pick a seed, mutate it multiple times to produce a set of test cases, T. Run the system under test with each of the cases in set T. For each input trace the coverage and monitor for crashes. If a test case is finding new coverage, add the seed to the seed queue. Repeat from step 2

According to my knowledge, all the coverage-guided fuzzers are using at least one of the following coverage metric: edge coverage, basic block coverage, path coverage. Basic blocks are sequential instructions which have the specific property that if one of them is executed, then every instruction in that block is executed. Edges are the branches and the decision points that might be visited during execution. The path is referred to as all the possible execution paths that can be taken while the program is executed. Line coverage is another metric that tests whether each line of the code has been executed or not. No fuzzer was found to be using line coverage.

1.2 Problem Definition

The goal of the Malware Analysis Lab project is to implement a coverage-guided fuzzer in Python when the source code is readily available. The fuzzer to be implemented should be mutation-based that accepts a valid input for the target system and generates new test cases by applying different types of mutations and code coverage analysis. The target for the fuzzer in a real-world program is limited to command-line arguments and standard input for C/C++ source code programs

1.3 Related Work

In the introduction, we discussed some key challenges that still need to be addressed to build an efficient coverage-guided fuzzer. There have been various previous works that have tried to provide a solution to the said challenges. In this section, we discuss some of the famous works.

The first challenge that needs to be addressed is improving test case generation. The importance of generating quality test cases has previously been proven. They can help improve the efficiency and effectiveness of the fuzzer. If compelling test cases are chosen, they can cover more code and prevent wastage of testing resources as ineffective test cases are discarded. The test cases can be chosen to specifically target a location in the code and hence, discover bugs quickly. The state-of-the-art coverage guided fuzzers like AFL [1] and libFuzzer [2] randomly mutate the input and blindly generate the test cases. The disadvantage of this method is low code coverage as in complex programs some checks may fail. There have been studies to address this problem from various directions. Fuzzers like Driller [18] and QSYM [20] use concolic execution to overcome sophisticated sanity checks by solving symbolic path constraints.

Another approach to overcome low coverage problem by making mutation smart by choosing where to mutate and how to mutate. Fuzzers like AFLFast [6] and Vuzzer [17] use static analysis to extract values affecting the block coverage before a fuzz run. The test cases that cover new blocks are identified and retained, and rest are discarded.

The next challenge that needs to be addressed is exploring the program code. Coverage based fuzzers use code coverage information to make the fuzzing efficient. The well-known fuzzers like libFuzzer [2] and Vuzzer [17] utilize block based coverage to improve their fuzzing techniques. AFL [1] on the other hand tracks edge coverage.

In [16] it was mentioned that line coverage is inefficient because the count of lines in complex programs can be exceptionally high and hence, difficult to track. According to [9] block coverage provides less information than edge coverage. Therefore, libFuzzer [2] and Vuzzer [17] are trying to improve their techniques by shifting to edge coverage. The technique of edge coverage, used by AFL[1] is not perfect as 75% of the edges can be untouched in complex programs due to the hash collision.

After carefully reading through some previous works, we observed that there were some issues at hand that had not yet been addressed. First, we noted that no work considered whether the chosen method of mutating inputs by mutation based fuzzers could be a deciding factor in improving fuzzer's efficiency. Second, no empirical study has been done to prove that line coverage is inefficient as compared to the more popular block coverage used in coverage based fuzzers. These observations inspired us to conduct these studies while implementing the given task.

1.4 Objective of the Project

After implementing the coverage-guided fuzzer as defined in problem definition, the objective was to study the open issues mentioned in the previous section. In this report, we formulate two hypotheses and use the implemented fuzzer to conduct experiments on the hypotheses. Also, the goal is to discuss the results and suggest further improvements.

Research Hypothesis 1: It is hypothesized that the type of mutation method chosen by a mutation based fuzzer, affects the number of bugs discovered in a particular amount of time and hence, improve the effectiveness of the fuzzer.

Research Hypothesis 2: It is hypothesized that line coverage is efficient as compared to block coverage and can help improve the efficiency of coverage based fuzzers by discovering more bugs in a stipulated amount of time.

2 Implementation

2.1 Fuzzer Solution

In this section, we describe the implementation phase the fuzzer architecture was proposed and developed. Due to the limited time, the goal was not to create a groundbreaking fuzzer but, to develop a fuzzer that can give decent coverage, is easy to understand and can be further extended. We named the fuzzer as *ItsSoFuzzy*. Figure 2 shows the proposed architecture of *ItsSoFuzzy* The following were the phases in which the fuzzer was developed:

2.1.1 Target Identification

With reference to the problem definition, the user should have the facility to test programs with either command line arguments or standard input. Therefore, an argument was added to the fuzzer, where the user needs to mention the type of program they want to test. *ItsSoFuzzy* can test programs that accept string inputs.

The purpose was to discover vulnerabilities like buffer overflows, use-after-free, double free and many more. To detect memory bugs easily, we decided that the input program should be compiled with AddressSanitizer which detects memory bugs in the Linux kernel.

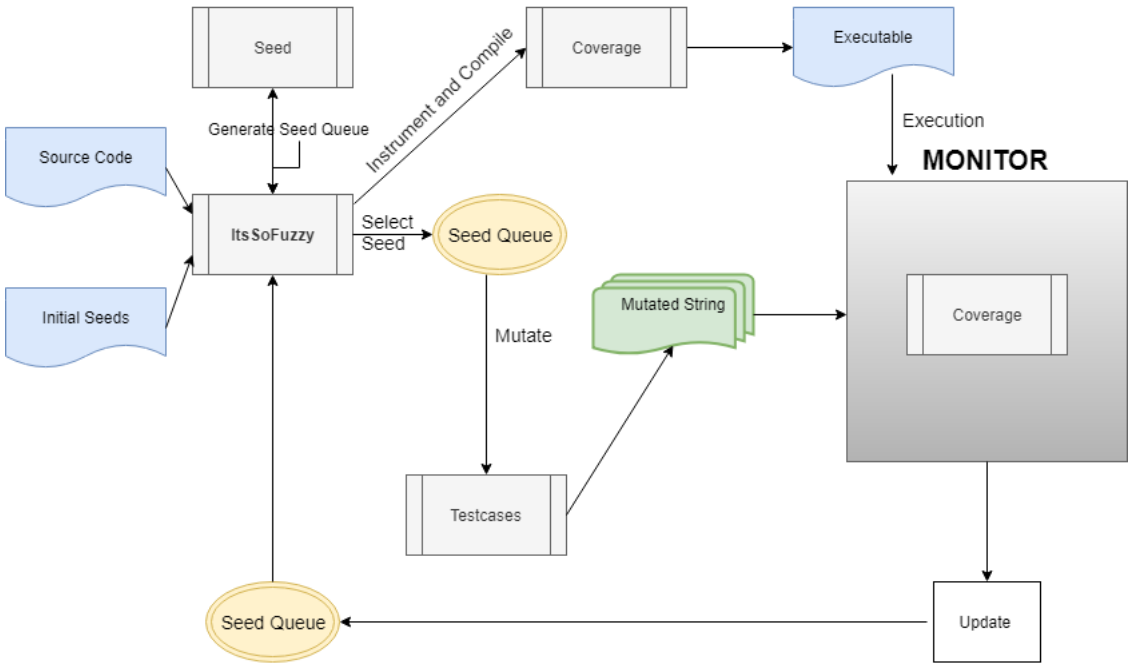


Figure 2: Proposed Fuzzer Architecture Diagram

2.1.2 Input Indentification

Using static analysis, the user should locate the potential sources of input and learn the expected input values. The format of acceptable input values is particularly important as that will help in creating the initial seeds. As discussed in the introduction, quality initial seeds can make testing effective and thus, a failure to locate the input requirements can indirectly affect the testing.

When the initial seeds have been decided upon, the user should provide the initial seeds in a file. If the file provided is empty, then *ItsSoFuzzy* generates ten random initial seed values to populate the seed queue.

2.1.3 Fuzz Data Generation

After populating the seed queue, the next step was to generate the fuzzed data. The goal of the project stated to use a mutation-based strategy. We have considered the state-the-art-fuzzer, AFL[1], as a reference and implemented some of its popular mutation methods as mentioned in [14]. We reached this decision after assessing that it is widely accepted as the most popular fuzzer.

The following mutation strategies were implemented:

Flip Bits: This technique [14] is very popular among mutation-based fuzzers like AFL[1], GPF[8], Radamsa[11] and zzuf[12]. It involves flipping a predefined number of bits or flipping a chosen number of bits. For this technique, I decided to use a predefined number of bits to keep it less complicated. *ItsSoFuzzy* has the functionality to flip one bit, flip two bits or flip all bits.

Byte Arithmetic: It is [14] another popular technique from AFL[1], it takes a random byte and adds a value r in the range of -35 ¡= r ¡= 35. The updated byte is replaced in the seed. The value of r is chosen at random and added to the byte sequence. The same functionality was added to *ItsSoFuzzy*.

Byte Swap: In this technique from Radamsa[11], bytes at random position are chosen and swapped. In *ItsSoFuzzy*, we implemented this technique in a way that every two bytes in the seed value are swapped.

Random: *ItsSoFuzzy* comes with the option to choose random as a mutation option in which before each execution of the program, the fuzzer randomly picks from the strategies mentioned above and mutates the seed value.

The initial idea was to generate an exhaustive set of test cases that included every possible mutation of a seed value. However, after failing to execute this fuzzer methodology due to system constraints, this idea was dropped.

According to [7], fuzzing is profoundly affected by the mutation ratio, and for every program, the perfect mutation ratio will be different. There were several previous papers to find the perfect mutation ratio. In [3] as they increased the number of iterations in mutation ratio, fuzzing proved to be more productive. After investigating the papers and considering the hardware and time constraints, we decided to fix the number of iterations at 10000.

2.1.4 Execution with Fuzzed Data

This step coincides with the previous step. This step needed to be automated because, without it, we are not fuzzing. After performing a mutation on the seed, *ItsSoFuzzy* compiles the program with necessary flags and executes the program. The flags used to compile the program are *fsanitize=address*, *–coverage*, *-ftest-coverage* and *-fprofile-arcs*. These flags provide necessary instrumentation to the program so that it is easier to trace the program behavior.

The user can provide the number of iterations for fuzz runs as *ItsSoFuzzy* has an optional argument where this information can be provided.

2.1.5 Exception Monitoring

This is one of the most critical steps in fuzzing. Even if we can crash a program with random inputs, it is of no use until we can tell which input caused the crash [14].

ItsSoFuzzy considers a successful run if the return code is 0 and considers an unsuccessful termination if the return code is -1. For all other return codes, it is considered that the program crashed. If the program is returning different return codes for a successful and unsuccessful run, then fuzzer will fail to provide accurate results. It is required that the user considers these details when testing a program.

If the execution results in a crash, then *ItsSoFuzzy* writes the input string and return code to an output file. The user can also define the name of this file as an argument.

2.1.6 Update Seed Queue using Coverage

This step was one of the key focus areas while implementing the project. This step could make the fuzzer more effective by adding only quality test cases to the seed queue. We achieve this by desiging a coverage-guided fuzzer. We trace the coverage for each fuzz run and analyze it to update the seed queue.

One of the main challenges was to choose the right tool to gather coverage information. There were conditions to be met before deciding on the tool: the tool should be easily available, able to parse C/C++ code, communicate with the python script and generate separate reports for each coverage metric. After a thorough assessment, we decided to use GCOV as the tool to get coverage information. It is available as an open source tool to measure coverage and can be used in conjunction with GCC. As stated earlier, the program is compiled with the proper flags that provide static instrumentation to the program. GCOV leverages this instrumentation to generate coverage reports that can be utilized further. GCOV can generate line, branch and block coverage reports.

ItsSoFuzzy uses the line and block coverage reports generated by GCOV. If a test case generates a new coverage that has not yet been seen, then it adds the test case to the seed queue. The test case is discarded in other scenarios.

Figure 3 and Figure 4 show the working flow of *ItsSoFuzzy*.

Figure 3: Fuzzer Algorithm Flow Diagram

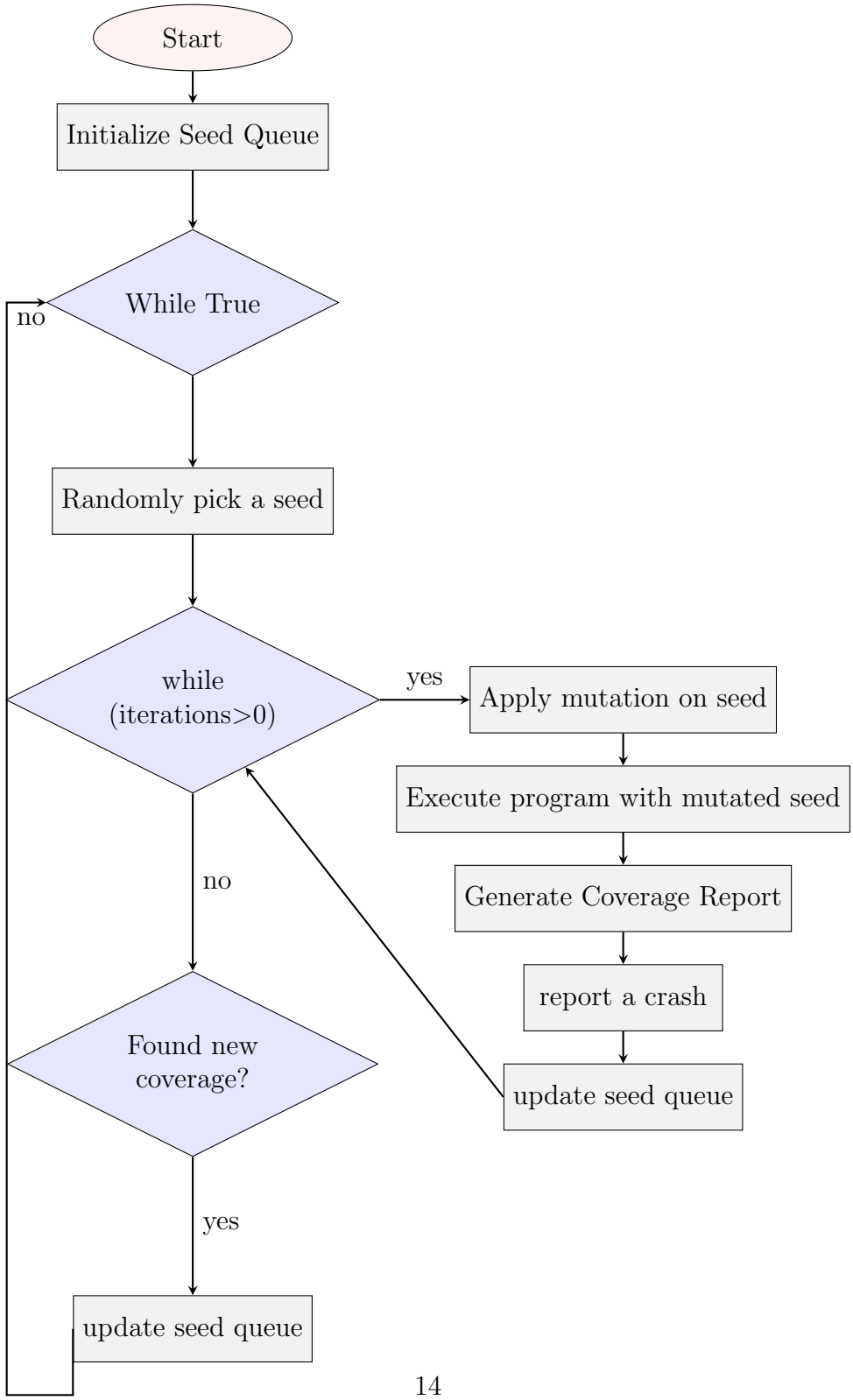
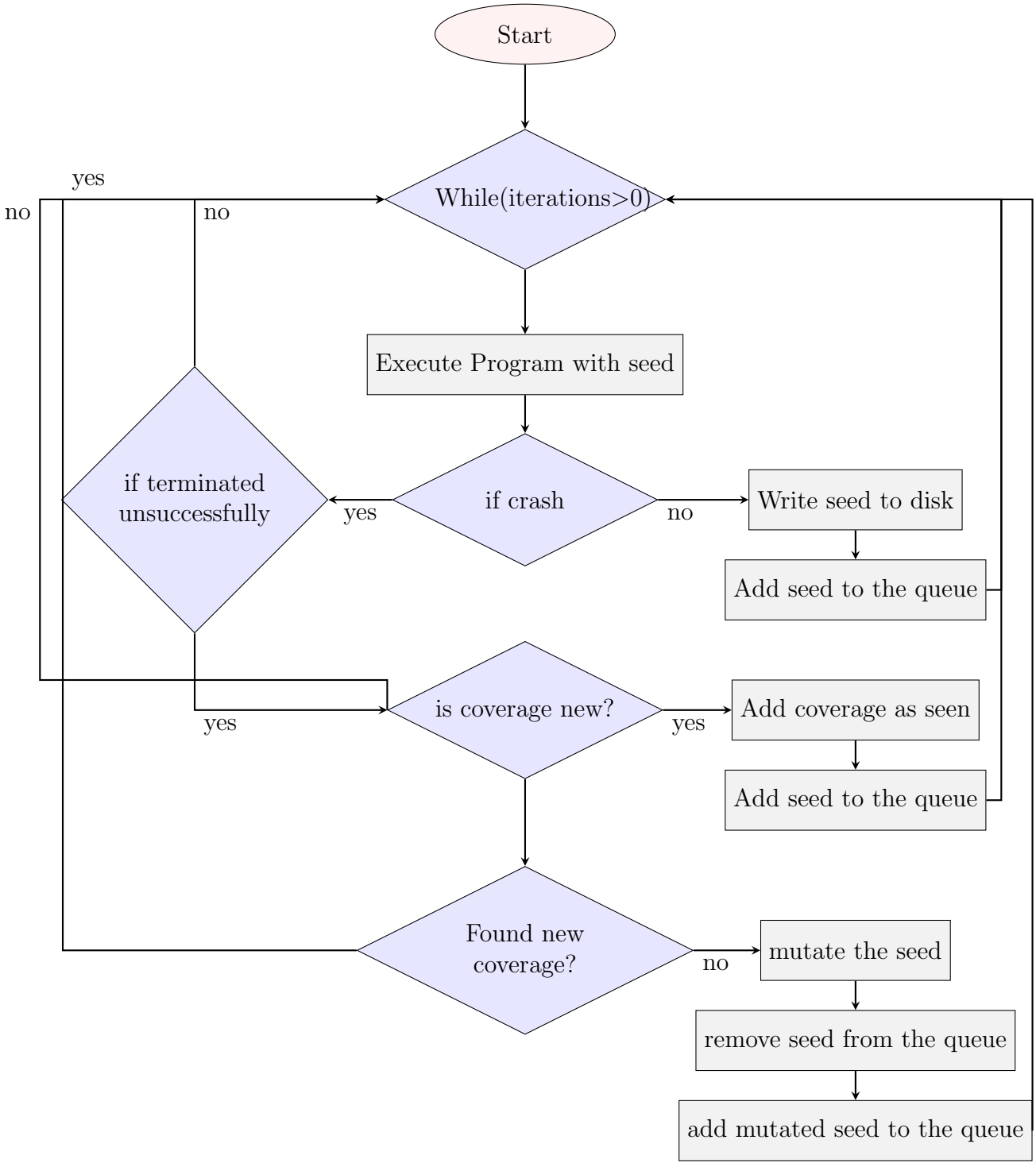
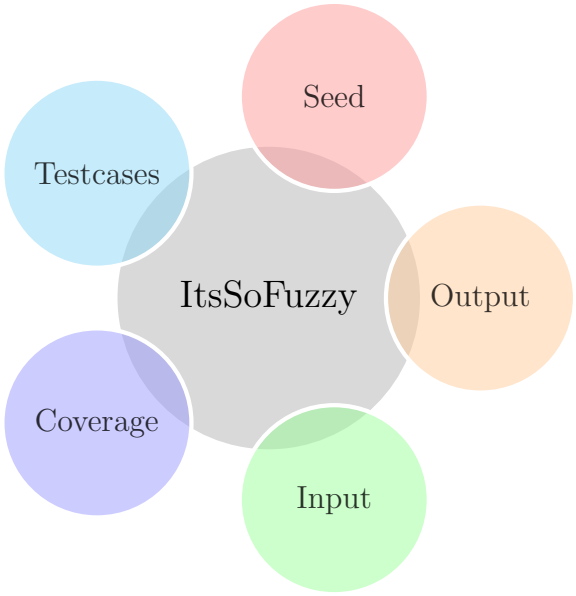


Figure 4: Update Seed Queue Flow Diagram



2.2 Modules



2.2.1 Seed

This module is used only when there are no intial seeds provided to the fuzzer. It has two functions. The first function returns a seed value generated with printable characters. The second function returns a seed value generated with all characters

2.2.2 Testcases

This module helps to mutate the seed value and generate a set of test cases. It accepts a seed value as input and returns the mutated seed. It contains two files implementing the mutation strategies. The strategies are divided among the two files depending on whether they are a bit-based mutation or a byte-based mutation.

2.2.3 Coverage

This module is responsible for compiling the program, generating coverage reports and reporting coverage of each test case. It also has a function that cleans the *.gcda* and *.gcno* files after every execution. This is necessary because the goal was to track coverage of each input and a failure to clean these files would have lead to a cumulative coverage report.

2.2.4 Input

The user is expected to place the program under test under this folder.

2.2.5 Output

The fuzzer generates an output file containing the information on crashes and places it under this folder.

2.2.6 ItsSoFuzzy

This is a python script that contains the fuzzer algorithm. It interacts with all the above mentioned modules to complete the fuzzing process. It accepts inputs from the user, populates the seed queue, executes the program under test and monitors for exceptions. It also analyzes the coverage of each test and updates the seed queue.

3 Evaluation

3.1 Evaluation Strategy

Our interest in this project was to experimentally evaluate the stated research hypothesis by designing experiments with the following goals:

1. Measure the number of crash inputs B produced by a certain pair of mutation method and coverage method (M, C) for a fixed number of iterations.

2. Measure the average of the crash inputs B produced by (M,C) across predetermined amounts of time.

3. Infer the effectiveness of the fuzzer depending on the average of crash inputs B collected by each mutation method and coverage method.

4. Deduce which mutation method increases the effectiveness of fuzzer.

5. Deduce which coverage method increases the efficiency of fuzzer.

The rest of this section describes the experiments. The evaluations were being carried out on a limited sample, therefore, the results needed to be gathered with caution.

3.2 Benchmark Programs

In this section, we discuss the programs that were put under test. The objective was to fuzz C/C++ programs that accept command line arguments or standard input. There was no standard test suite available for these requirements and therefore, we either coded the program or chose projects [10] and [5] that required slight modification to meet our requirements.

The programs are chosen based on: length, complexity and diverse category. We evaluated 3 programs that are categorized according to complexity as: small, medium and large. The small program is a manually-written test-suite, while the medium and large programs are projects with manually-induced vulnerabilities.

3.3 Experimental Setup

We used the coverage-guided fuzzer with the following configurations and then compared them to deduce results.

1. Flip-one and line coverage (F, L)
2. Byte-arithmetic and line coverage (By, L)
3. Random and line coverage (R, L)
4. Flip-one and block coverage (F, B)
5. Byte-arithmetic and block coverage (By, B)
6. Random and block coverage (R, B)

Due to the shortened scope and time of the project, rest of the mutation strategies were not compared.

Performance measures: For our evaluation, we measured the number of crashes found by the fuzzer over a period of time, where inputs were guided by coverage. As crashes are an indication of potentially serious bugs, choosing this metric to test the fuzzer seemed natural. The number of lines and blocks covered were also noted so as to compare the two coverage metrics. Higher coverage of the program can lead to uncovering of more bugs.

Platform: The experiments were conducted on one machine. The machine came equipped with 16 Intel(R) Xeon(R) Gold 6130 2.10GHz CPUs and 32 GB RAM with Ubuntu 18.04. *ItsSoFuzzy* script took advantage of the multiprocessors to run as many tests as possible in parallel. Every configuration was allowed to run for increasing amounts of time and we measured

30 tests per configuration. The testing was done with a set of valid input seeds.

Seed Selection: All the configurations to be tested required a file with initial seeds. It was known that the length of the inputs could affect the testing. To select the appropriate length, inputs of different lengths and formats were tested. The inputs that provided different coverages were taken as initial seeds. By analyzing further, 15-20 inputs were used for the programs. The same set of inputs was used to test each of the configurations on a program.

3.4 Results

Table 1, Table 2 and Table 3 summarizes the crashes and coverage results for small, medium and large programs respectively. The average number of crashes and average coverage achieved was calculated over 30 test runs. The minimum time to test a configuration was 5 minutes and the maximum time was capped at 2 hours and 30 minutes.

Table 1: Small Program Results

	Fuzzing Strategies					
Results	(F,L)	(By,L)	(R,L)	(F,B)	(By,B)	(R,B)
Total Crashes	297325	350897	544776	321077	365782	554894
Avg Inputs	24344	24777	24388	24256	24410	25613
Avg Crashes	9910	11697	18159	10702	12192	18496
Avg Coverage	66.67%	66.67%	66.67%	100%	100%	100%

Table 2: Medium Program Results

	Fuzzing Strategies					
Results	(F,L)	(By,L)	(R,L)	(F,B)	(By,B)	(R,B)
Total Crashes	494441	559238	561128	680618	710214	737798
Avg Inputs	29787	29762	29190	30654	29865	30127
Avg Crashes	16481	18641	18704	22672	23673	24593
Avg Coverage	8.9%	9.01%	10.27%	20%	21.4%	21.7%

Table 3: Large Program Results

	Fuzzing Strategies					
Results	(F,L)	(By,L)	(R,L)	(F,B)	(By,B)	(R,B)
Total Crashes	408952	426788	426861	487653	513471	513440
Avg Inputs	22113	22432	25145	25644	25543	25532
Avg Crashes	13619	14266	14268	16255	17114	17115
Avg Coverage	44%	48.5%	50.5%	52.7%	55.6%	55.9%

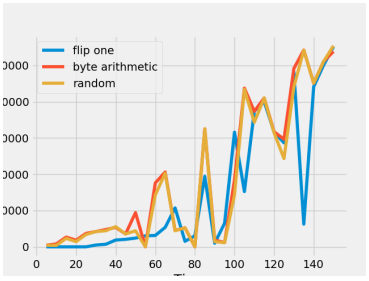
From the average crashes result, we assume that random mutation strategy has a slight edge over the other mutation strategies considered. This behaviour seems to be consistent across all the three programs. For small program, (R,L) configuration results in higher average number of crashes even when the average number of inputs provided were lesser as compared to (By,L) configuration. For large program, the random mutation seems to have a very small advantage over other mutations. Flip-one mutation strategy seems to be the least-performing in each section.

Figure 5, shows the growth of the number of crashes for each mutation strategy for all the three programs. Flip-one mutation strategy appears to be slow in getting higher number of crashes. The general trend of number of crashes over time is observed to be increasing. It is worth noting that the number of crashes for each strategy seem to be converging with increasing time.

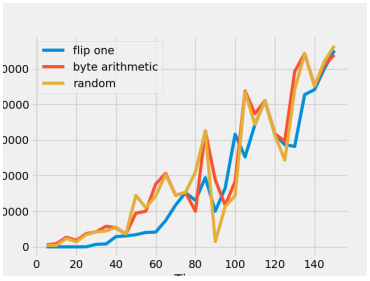
Code-coverage is the one of the important features of coverage-guided fuzzers. We measured the average coverage achieved by each configuration over 30 runs. Comparing the results, it appears that block coverage provides considerably higher coverage for small and medium programs. For large programs too, block coverage appears to be the winner but the difference in the average achieved coverage is not too big. There also appears to be a clear gap in the number of crashes found with block coverage as compared to line coverage.

Figure 6, shows the growth of line coverage for the 3 programs. For small and medium programs, it is observed that random mutation achieves a high coverage before the other strategies. Random mutation seems to be outperforming others on the coverage achieved in all the 3 programs under test.

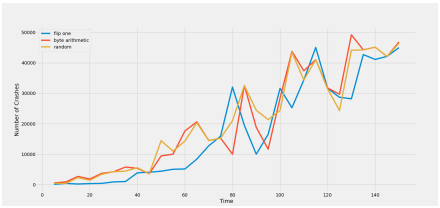
Figure 7, shows the growth of block coverage for the programs under test. It appears that in the small program none of the strategies had any particular advantage over the other as all have the same block coverage growth. In the other two, random strategy is touching the highest coverage in the graphs.



(a) Small Program

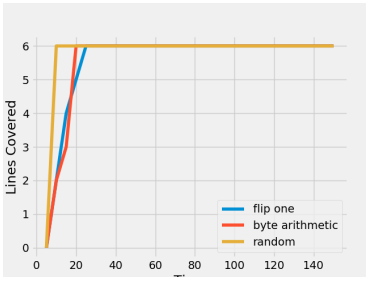


(b) Medium Program

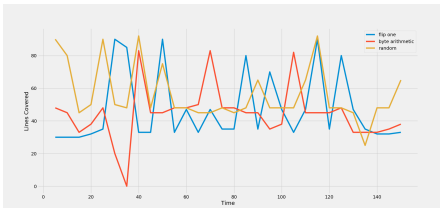


(c) Large Program

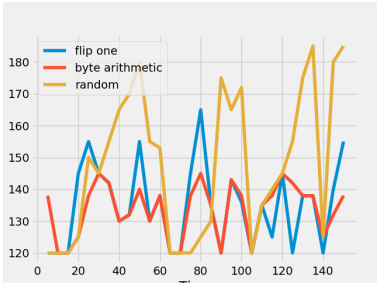
Figure 5: Crashes Over Time



(a) Small Program

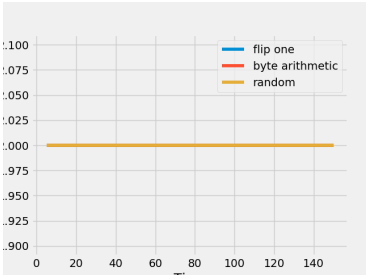


(b) Medium Program



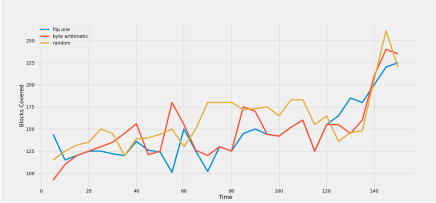
(c) Large Program

Figure 6: Lines Covered Over Time



(a) Small Program

(b) Medium Program



(c) Large Program

Figure 7: Blocks Covered Over Time

4 Discussion

This section discusses the hypotheses formulated in the beginning.

4.1 Research Hypothesis 1

As defined in Section 1.4, the idea was that the mutation strategy could affect the fuzzer effectiveness. We assessed the results Among the strategies considered, random mutation performed better than the others in the stipulated amount of time. It had higher average number of crashes and average coverage achieved. Flip-one mutation was the least performing strategy. We assume, it shows that choosing the correct mutation strategy might affect the effectiveness of the fuzzer. The formulated hypothesis appears to be correct.

4.2 Research Hypothesis 2

The idea formulated was that line coverage can prove to be better coverage metric than block coverage for coverage-guided fuzzers. With respect to the results obtained, in all the cases block coverage outperformed line coverage. It was also noted that the average number of crashes obtained with block coverage as the metric were considerably more than line coverage. The formulated hypothesis appears to be incorrect.

5 Conclusion and Future Work

During the lab, a coverage-guided fuzzer was implemented in Python. In this report, we hypothesized that the choice of mutation methods can affect the fuzzing results and also, that line coverage could be an effective coverage metric to use in fuzzers. With the help of the implemented fuzzer these hypotheses were experimentally evaluated. The results were assumed to be slightly favorable towards using random strategy for mutation and block coverage as coverage metric. It should be noted that these experiments are just a groundwork on the ideas.

We saw three important lines of future work. First, there is a need to develop a standardized benchmark suite to test the fuzzers. Also, *ItsSo-Fuzzy* can be extended to handle infinite loops, testing of binaries etc. Most importantly, more rigorous testing is required to solidify the ideas in this report.

6 Acknowledgements

We would like to thank our supervisor Mr. Daniel Baier for supporting our work and help get results of better quality. We would also like to express our gratitude to Mr. Martin Clauß, Research Assistant, for his patience and support through numerous challenges faced during this project.

References

[1] M. Zalewski "American Fuzzy Lop" [Online] Available: http://lcamtuf.coredump.cx/afl/.

[2] K. Serebryany, "Continuous fuzzing with libfuzzer and address sanitizer," in IEEE Cybersecurity Development Conference, ser. SecDev,2016, pp. 157–15.

[3]

[4] Omar Alhazmi and Yashwant Malaiya. Modeling the vulnerability discovery process. *2010 IEEE 21st International Symposium on Software Reliability Engineering*, 0:129–138, 11 2005.

[5] Max Base. Decodequerystringc. https://github.com/BaseMax/DecodeQueryStringC.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1032–1043, New York, NY, USA, 2016. ACM.

[7] Sang Kil Cha, Maverick Woo, and David Brumley. Program-adaptive mutational fuzzing. In *2015 IEEE Symposium on Security and Privacy*, pages 725–741. IEEE, 2015.

[8] Jared Demott, Dr Richard, R.J. Enbody, Dr William, and William Punch. Revolutionizing the field of grey-box attack surface testing with evolutionary fuzzing, 04 2019.

[9] Shuitao Gan, Chao Zhang, Xiaojun Qin, Xuwen Tu, Kang Li, Zhongyu Pei, and Zuoning Chen. Collafl: Path sensitive fuzzing. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 679–696. IEEE, 2018.

[10] Bart Grantham. Qs parse. https://github.com/bartgrantham/qs$_parse$, 2010.

[11] Aki Helin. Radamsa. https://gitlab.com/akihe/radamsa.

[12] Sam Hocevar. Zzuf. https://github.com/samhocevar/zzuf.

[13] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, Jun 2018.

[14] Valentin J. M. Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J. Schwartz, and Maverick Woo. Fuzzing: Art, science, and engineering. *CoRR*, abs/1812.00140, 2018.

[15] Stefan Nagy and Matthew Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing, 12 2018.

[16] Saahil Ognawala, Fabian Kilger, and Alexander Pretschner. Compositional fuzzing aided by targeted symbolic execution. *arXiv preprint arXiv:1903.02981*, 2019.

[17] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. Vuzzer: Application-aware evolutionary fuzzing. 2017.

[18] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In *Proceedings of the Network and Distributed System Security Symposium*, 2016.

[19] Michael Sutton, Adam Greene, and Pedram Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[20] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th {USENIX} Security Symposium ({USENIX} Security 18)*, pages 745–761, 2018.