Generating Tests for the Discovery of Security Flaws in Product Variants

Francisco Araújo, Ibéria Medeiros, Nuno Neves
LASIGE, Faculdade de Ciências, Universidade de Lisboa, Portugal
fc45701@fc.ul.pt, imedeiros@di.fc.ul.pt, nuno@di.fc.ul.pt

***Abstract*—Industrial products, like vehicles and trains, integrate embedded systems implementing diverse and complicated functionalities. Such functionalities are programmable by software and contain a multitude of parameters necessary for their configuration, which have been increasing due to the market diversification and customer demand. In addition, industrial products are often built by aggregating different software parts (components), constituting thus *product variants*. Product variants with such variability need to be tested adequately, in particular if one is concerned with security vulnerabilities. While efficient automated testing approaches already exist, such as fuzzing, no tool is able to use results from previous testing campaigns to increase the efficiency of security testing the next product variant that shares certain functionalities. This paper presents an approach that can ignore already covered functionalities by previous tests and give more importance to blocks of code that have yet to be checked. The benefit is to avoid repeating unnecessary work, hence increasing the speed and the coverage in the new variant. The approach was implemented in a tool based on the AFL fuzzer and was validated with a set of programs of different versions. The experimental results show that the tool can perform better than AFL in our testing scenario.**

***Index Terms*—Fuzzing, Vulnerability detection, Coverage testing, Software Variant testing, Software security**

I. Introduction

The swift growth of software complexity coupled with the demand for many software-based products in everyday life activities, has caused a significant need for efficient software testing approaches to assure certain security and quality standards. As an example, a car requires extremely complex and critical software which, when put under stress, could eventually activate some sort of bug. If any of the existing bugs happens to be exploitable (i.e., it is a vulnerability), then it could give an opportunity to an external attacker to control the car, putting lives in danger and originating millions in damages. In addition, correcting the bug would incur in high costs as updating the software in already sold vehicles has induced losses in the order of billions annually.

Many of the industrial software products suffer from being highly complex and diverse, requiring large development teams, and as such they can end up having a fair amount of flaws. In fact, open statistics show an important number of vulnerabilities being found each year in very diverse software [1]. Such vulnerabilities can stay undetected for considerable periods, since there are often pressures that limit the testing period and security bugs are particularly difficult to find in an automated manner. Attackers, on the other hand, have a theoretically infinite time to discover those vulnerabilities and only need to find one to compromise the system and run whatever malicious code they desire. As a result, most companies nowadays spend an increasing amount of time, money and expertise in software testing and verification. However, as previously explained, since resources are always limited, many bugs still manage to remain in the final products.

In an industrial piece of software, there are usually several code blocks or modules that can be reused in other products. The inclusion or not of shared code also occurs depending on configuration parameters, when setting up distinct functionalities in a product line, creating in fact various versions of a program. In addition, third-party software components can be present in many different programs. Software re-usability brings many benefits, such as allowing a faster ensemble of new software. However, it also lets bugs to be carried and to be inserted all product variants[1] that happen to utilize them [2], and so it can lead to an increase in the number of vulnerabilities.

Nowadays, fuzzing is probably the most effective state-of-the-art software testing approach for vulnerability discovery. It works by feeding the program with randomly-generated inputs, recording any crashes found while doing so. It has been used successfully by major software companies for security testing and quality assurance. However, no fuzzer to date takes into consideration already tested modules or code blocks shared between product variants. As such, a lot of redundant tests occur, which decreases the efficiency of the procedure. This happens because tools are busy exploring already checked program paths, eventually finding equivalent flaws, with not much added value to the company conducting the tests.

The paper presents a greybox fuzzing approach for detecting vulnerabilities in variants developed in the C/C++. The technique was implemented in a fuzzer called `PandoraFuzzer`, which was constructed by modifying the AFL fuzzer [3], [4]. The resulting fuzzer can find software flaws in a simple, efficient and productive way, as it is common among fuzzers, but without having to redo the testing of already covered functionalities, or modules, that are common among product variants. This is achieved by retaining the results of previous tests to keep track of the code blocks already tested. As such, it can minimize the amount of repeated test cases done

[1] In the rest of the paper, we will call software that shares modules or components as *product variants* or simply *variants*.

and maximize the coverage on the unshared functionalities, saving precious time and money. In our initial experiments, we have resorted to different program versions as a way to get variants. The results demonstrate that `PandoraFuzzer` can potentially perform better than AFL, in a testing scenario where multiple product variants have to be validated.

The contributions of the paper are: (1) A greybox fuzzing approach that can be used to detect vulnerabilities in multiple program variants. The proposed architecture takes advantage of previously done test procedures to best understand how to test a given variant without redoing work. (2) The `PandoraFuzzer` tool that shows: (i) the capacity to learn from previous fuzzing efforts, avoiding/reducing the generation of test cases that check already evaluated code parts; (ii) the ability to replicate the crashes found during fuzzing by providing to the user the test cases that crashed the programs. (3) An experimental evaluation of `PandoraFuzzer` with four large applications from the GNU binutils, to reveal the viability and effectiveness of the tool when compared to AFL.

The outline of this paper is as follows. Section II explains some relevant concepts and provides fundamental context for the work done in this paper. Section III is dedicated to explaining the proposed solution and issues it addresses, and Sections IV and V detail its main components. Sections VI and VII describe the current implementation of the proposed architecture in the `PandoraFuzzer` tool as well as evaluating and validating the tool with four applications. Section VIII concludes the paper.

II. Concepts and Related Work

A. Software Validation and Vulnerabilities

Software validation is the process of making sure the software matches the needs of the user and works as intended, fulfils the software requirements and specifications and has the least number of bugs possible. Each test examines the behaviour of the software under test (SUT) to verify if it has any incoherent behaviour and contributes to raise the confidence on the correctness of the product. Testing often works by developing inputs that have a certain code coverage of the program. This is usually done with knowledge of the program to be tested. In order to increase the speed and the amount of testing done, automation of software verification should be performed whenever possible. In software validation, a test case is composed of the test values and the expected results, and a test set is a set of test cases.

Testing is made to ensure three important features of SUT: (1) software functionalities: testing the functionalities for what SUT was designed; (2) safety: testing if the safety properties of SUT are being ensured; (3) security: testing if SUT does not contain vulnerabilities which can compromise its the correct behaviour and inccur in software failures. For safety and security, an important distinction must be made between a fault/bug, an error and a failure. A software fault is a defect. An error is an incorrect state of the program, for example, an invalid value in a variable that happens due to some fault. A software failure is an incorrect behaviour with respect to the requirements defined for the program. Another distinction that should be made is between bug and vulnerability. A vulnerability is a bug, but not all bugs are vulnerabilities. A vulnerability can be described as a flaw or weakness in the application which can be the result of a design flaw or a simple implementation bug, which allows an attacker to exploit it to compromise the security properties of an application.

Vulnerabilities are the root cause of security problems and when they are exploited by attackers, they can cause damage to the system. In this paper, we consider the classes of vulnerabilities identified in the Common Weakness Enumeration (CWE) [5] as problems for the applications programmed in the C and C++ languages, namely *variable overflow and underflow*, *integer* and *memory*. For the former, for example, HOTracer finds heap overflows vulnerabilities by using dynamic analysis [6], and angr does vulnerability detection through a binary analysis [7]. For integer vulnerabilities, SwordFuzzer [8] and DEEEP [9] use taint analysis to discover such bugs. To detect vulnerabilities related to memory leaks, valgring [10] and LAVA [11] are two tools for the effect. Muench et al. [12] and Veen et al. [13] also investigated this class of vulnerabilities.

B. Fuzzing

Fuzzing is a popular technique for finding software bugs where the SUT is barraged with randomly generated test cases. It is used for security testing and quality assurance proposes, such as in the work done by Takanen [14], and by Oehlert [15], ever since it was introduced. While the program is being put under test, it is monitored in the hopes of finding errors that might arise as a result of the input given.

Although fuzzers differ in many significant ways, in general, most of them follow the algorithm shown below.

```
1 General Fuzzing Algorithm{
2   queue <- Initial Test Cases of concrete valid input
3   while ( not DoneWithFuzzing ){
4     chosenTestCase <- chooseTestCase(queue)
5     runProg(chosenTestCase)
6     mutatedTestCase <- mutate(chosenTestCase)
7     if (isInteresting(mutatedTestCase))
8          queue <- addToQueue(mutatedTestCase)
9   }
10 }
```

The algorithm always receives, and returns, concrete valid inputs (or test cases) that the SUT processes. After running the program with the received input, it mutates the input used in the execution to generate new input, which might lead to different paths being covered when it is executed. Some fuzzers also use the information gathered in the execution to help generate and pick better program inputs. If the program input is deemed interesting, it is saved to the queue in order to be further mutated to uncover different paths in the program. In the end, it is necessary to decide if the fuzzing process is done. This is generally accomplished by a timeout or by reaching a certain number of discovered bugs, with the ultimate goal of trying to find inputs to make the project crash. These inputs are then returned to the developers and testers that can use them to locate the bug and reproduce the crash.

There are three categories of fuzzers: *blackbox*, *whitebox*, and *greybox*. Our work relies on the last one, which we present next the works we consider more relevant and related to our solution. However, we briefly give first an overview of the other two categories.

Blackbox fuzzers were the original fuzzers. They treat the program as a blackbox, i.e., without having any knowledge about the source code of the program. Even without having prior knowledge, they have to generate an instrumental amount of random test cases in a very short amount of time to perform the fuzzing task. They has the greatest ability to generate the largest amount of tests, but only provides limited coverage and so the testing can be very inefficient. FuzzSim [16] is an example of blackbox fuzzer.

Whitebox fuzzers fix many of the limitations blackbox fuzzers have, since this sort of fuzzers miss bugs that depend on specific triggers values. Starting from a well-formed input, whitebox fuzzing consists of symbolically executing the SUT dynamically, gathering constraints on inputs from conditional branches encountered along the execution. The most known whitebox fuzzer is SAGE [17]. Other works were developed based it, as the Bounimova et al. [18].

Greybox fuzzers uses only lightweight instrumentation to glean on the program structure without requiring any previous analysis. This may cause a significant performance overhead but increases the code coverage as a result. In practice, greybox fuzzing may be more efficient than whitebox fuzzing with more information about the internal structure of a program and it may also be more effective than blackbox fuzzing.

Hawkeye [19] combines static analysis and dynamic fuzzing for finding C/C++ vulnerabilities. VUzzer [20] is a fuzzer that implements a feedback loop to help generate new inputs from the old ones, with its two main components being a static analyser and a dynamic fuzzing loop. S. Karamcheti et al. [21] show that sampling distribution over mutational operators can improve the performance of AFL. They also introduce Thompson Sampling, which is a bandit-based optimization to improve the mutator distribution adaptively. They focus on improving greybox fuzzing by studying the selection of the most promising parent test case to mutate. LibFuzzer is a coverage-guided, evolutionary fuzzing engine to test C/C++ software [22]. Another such fuzzer is honggfuzz [23], a security oriented, feedback-driven, evolutionary fuzzer. Alexandre et al. [24] presented a way to optimize test case selection in order to increase coverage of the software under test. Grieco et al. [25] developed VDiscover, a tool to predict if a test case is likely to discover software vulnerabilities by using lightweight static and dynamic features implemented using machine learning techniques. Klees et al. [26] propose some guidelines to better test and evaluate fuzzing algorithms.

Despite there are various fuzzing works that address C/C++ vulnerabilities, none of them takes into account the results of variant tests already made for a product to be reused in other product variant. Also, they do not learn such tests in order to prioritize them when reuse them. The approach we propose contains such features.

C. AFL

AFL (American Fuzzy Lop), is one of the most popular and used greybox fuzzers. A fuzzer works by testing the software target by barraging it with test cases generated automatically through mutations. AFL can execute hundreds to thousands of inputs per second, covering a large amount of the program attack surface in a relatively short amount of time. In a broad sense, AFL selects a prior promising parent test case to sample, mutates its contents, and executes the program with the resulting child input. It verifies the behaviour of the target software against incorrect data inputs to find flaws, such as faulty memory management, assertion violations, incorrect Null handling, bad exception handling, deadlocks, infinite loops, and undefined behaviours. If such flaws are reached and exploited the SUT can crash and stay inoperable.

An AFL key is the use of coverage information obtained during the execution of the previously generated testing inputs, which is achieved by the injection of lightweight instrumentation in the SUT during compilation. More specifically, after the assembling stage has finished but before the linking stage has started, a few assembly instructions are added to each basic block to identify them and to track the path an input takes while being processed by the SUT. This is done because relying solely on random mutations decreases the chances to reach certain previously unseen parts of the program. The instrumentation has a modest performance impact and aims to identify new paths in the program and to have the ability to find the edges have been passed on the program.

Another key behind AFL is its forkserver component. The most common way to fuzz programs is to just keep executing the SUT over and over with different random inputs. This approach has its problems as most of the time might be spent waiting for program cloning (*execve*), the linker and all the library initialization routines, to do their jobs. To face to this issue, the forkserver' AFL lets *execve* happen, get past the linker and then stop early in the actual program, before it gets to process any inputs generated by the fuzzer. Once the SUT reaches the designated point in the program, it simply waits for commands from the fuzzer. When it receives a "go" message, then it calls the function *fork* to create an identical clone of the already-loaded program. The injected code returns control to the original binary, letting it process the fuzzer-supplied input data and then relay the PID of the child process to the fuzzer, In the end, it goes back to the command-wait loop.

After the instrumentalization is done, the fuzzing phase starts by passing through every interesting test cases, mutating them in order to find new interesting test cases, i.e., if they trigger new coverage. If the test case or any of its mutations cause a crash of the program, then the test case is added to the *crashQueue*. This queue keeps all the inputs that crashed the program. If the program times out, then the test case is added to the *hangQueue*.

III. PANDORAFUZZER GREYBOX FUZZER APPROACH

This section, before presenting the proposed approach, gives an explanation of issues and challenges it intends to address.

A. Issues and Challenges

All the issues and challenges enumerated below occurred with the intent to propose an approach that learns how to best test a given variant and also focuses the fuzzing efforts on the patch fixes. This must be achieved while still fuzzing the original program for any vulnerabilities that might have yet to be detected and might have made it to the patched application.

1) *Shared Functionality Discovery* - In order to direct the SUT to targets that have yet to be fuzzed, we need to build test cases that do not cause the execution of functionalities that have previously been fuzzed in the program, or that are shared between programs. Hence, there is the need to allow the fuzzer to avoid repeating work. This gives the tool the ability to reach for example a patch location, letting it fuzz the code modified by the patch sooner than AFL.
2) *Multiple Program Fuzzing* - One forkserver would not suffice to fuzz more than a single program. Hence the forkserver logic itself should be changed somewhat to allow for more than one program to be fuzzed at a time. Multiple program fuzzing allows for testing of both SUT, not only focusing the fuzzing efforts on the patched part of the program but also the unpatched region that might still have to be tested. This gives the solution the ability to find previously undiscovered vulnerabilities that perchance might have been passed into the next version of the program.
3) *Interesting Program Input Interchange* - In order to avoid repeating work while fuzzing, interesting program inputs that can trigger new behaviour in more than one program variant have to be shared among all testing operations, so they can learn from it. The solution to this problem allows hidden vulnerabilities or hidden paths in one variant to be discovered by the other program variants faster than if we only had been fuzzing each SUT separately.

B. Approach Overview

The goal of the proposed greybox fuzzer approach is to discover vulnerabilities in product variants, resorting of the test cases made between variants. As product variants contain software parts (e.g., modules, components) that can be common to different products, our approach proposes to reuse the test cases of a given software part that were already executed for a product in another variant. For one hand, this avoids repeating tests that have already been seen and allows reducing the spent time on testing products. For the other hand, this allows the approach to learn how to best test a given product variant.

The approach focuses on the development of a solution based on AFL, i.e., a modified AFL version that could solve all the issues described in the previous section. As such, it allows for a multitude of programs to be fuzzed at the same time and, for various crashes and interesting inputs to be saved for each and every one of the variants in such a way to facilitate the interaction with the user. In addition, it allows, for interesting program input interchange, supporting the sharing of previous tests that have been performed. Hence, it is expected a faster vulnerability discovery for example if bugs were introduced in a patch correction.

The approach comprises two phases, namely *Instrumentalization* and *detecting flaws*. The goal of the instrumentalization phase is to simply facilitate the second phase by providing information about the structure of the SUT, namely the identification of the basic blocks that compose it. As such, the detecting flaws phase, when fuzzing the various program variants, can decide which blocks to target in the future and which blocks are less interesting. The fuzzing is done by generation, execution and reusing test cases to detect flaws in SUT. The instrumentalization phase also allows for faster execution rate by means of the forkserver, explained in Section IV-A5. More details over these phases are described in the next two sections.

IV. Instrumentalization and Basic Blocks Identification

A. Instrumentalization

The instrumentalization phase comprises five modules which are described next. In Fig. 1 are represented the main steps a program goes through while being compiled by afl-gcc, resulting the assembly code of the program, which then is instrumentalized.

1) Initialize Instrumentalization: Performs the initialization of the instrumentalization by means of processing the program source code. It acts as a compiler, instrumentalizing the program while the executable (assembly code) is being generated.

2) Detect Basic Block: Goes through all basic blocks in the resulting assembly code from the provided source code, essentially instrumentalizing all basic blocks detected.

3) Identification of Basic Blocks: Structural information of any given program is highly important in our solution. The outputs of this module are used by many of the other modules

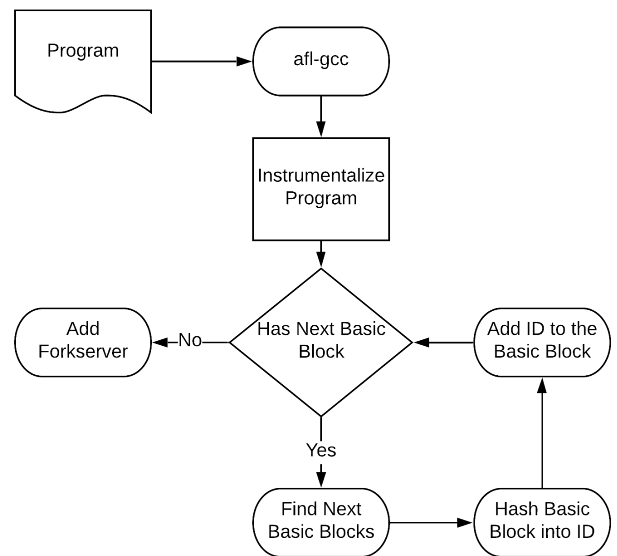

Fig. 1. Instrumentalization and basic blocks identification phase data flow.

from the fuzzing process. Nevertheless, this module aims to generate unique basic blocks identifiers that will allow the fuzzer to identify shared blocks between variants. To do so, the ID is the hash of the contents of the basic block itself.

This is the major difference between AFL and our approach, i.e., the way the basic block identifier is generated. In AFL it consists of simply creating a random value. This is enough in AFL case because it is not looking for shared structure information across program variants. However, our approach is looking for the ability to differentiate between two programs functionalities. This is done at the basic block level. Since the ID is based on the contents of the basic block itself, this means that, all different basic blocks will have a distinct identifier associated to them. As such, if two programs share any basic bock, they will share the same IDs. The process of generating IDs and an example is presented in Section IV-B.

4) Modify Assembly Code: Modification of the assembly code is done much like AFL in our approach. Simply adding the identifier to the basic blocks.

5) Forkserver: When fuzzing any given program, the simplest way to do it is to find any given test case that exercises the desired functionalities and then keep executing it over and over again. This, however, is not the optimal way of fuzzing any given application, since the tool needs to continuously repeat slow operations like the *execve* system call, the linking of all libraries, and all the library initialization routines.

As we stated previously, the forkerver is an injection of a small piece of code into the program being tested, with the goal to let *execve* happen, get past the linker and stop before the program starts processing any inputs. Once this is done, the forkserver simply waits for a *go* command, calls the function *fork*, and then creates an identical clone of the already-loaded program and continues processing the input. This mechanism does not change from the one provided by AFL, except for the support for multiple program variants.

B. Basic Blocks Identification

As we said before, a basic block is a sequence of code lines with no jumps in between. Compilers usually decompose programs into their basic blocks as a first step in the analysis process, so we take advantage of this feature to get these basic blocks and identify them uniquely. Nevertheless, we can identify which ones are shared between variants and were already tested.

The generation of the basic block identifier is made in three steps, which are presented next and illustrated in Fig. 2.

First step. The goal of this step follows the idea: to avoid having an identifier that depends on the content of a basic block that could change among program variants, we remove all unnecessary lines that do not affect functionality. To do so, firstly, the assembly code of any basic block is divided into lines, and then all unimportant lines from the assembly code (as indicated in red in the figure), such as line information or labels, are removed. *Second step.* This step removes the registers (e.g., *eax* as marked on red in the figure) because their contents are volatile. Hence, they could change across program variants. *Third step.* All the remaining lines are concatenated into a single line and its hash is calculated, generating thus the ID. The hash is an adaptation of the Pearson hashing to guarantee better distribution of the hash values and the fast execution on the registers of the processor.

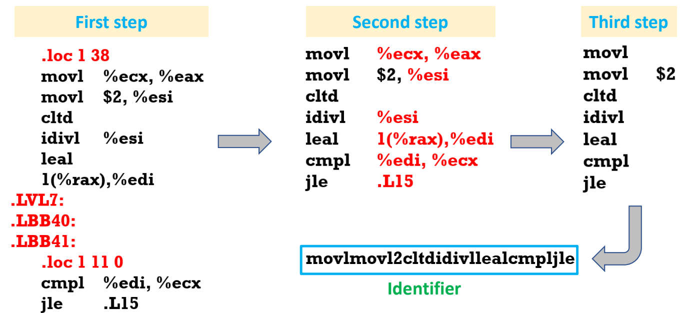

Fig. 2. Example of basic block identifier generation.

V. Detecting Flaws

The detecting flaws phase is responsible for fuzzing the SUT, reusing the tests results of the basic blocks shared between variants and already tested. It is composed of six modules which are illustrated in Fig. 3 and explained next.

1) Queue Handler: It is of the upmost importance the ability to maintain information about multiple test cases in an efficient manner. For that, the queue handler contains, manages and interacts with the queues that represent the exploration state of the program. Each variant will be represented by a Queue in the Queue handler. That means that each queue contains the various basic blocks belonging to the program, as well as the information about all interesting inputs to be fuzzed in the program. The queue handler also contains information about all top test cases for each program, where a top test case represents the best test case that can reach a specific basic block. A basic block is an element of the queue which belongs to an hash map. This map allows the Queue Handler to verify quickly if an element is already in the queue, to avoid repeating the work of initializing the element twice and to quickly add the said element to the queue. The basic block ID is used as the key element of the hash map and the value is the number of times it was executed during the fuzzing process. Hence, an element of the hash maps that as a value equal to zero represents a basic block which may or may not be in the program itself. This basic block has yet to be seen during the fuzzing process of the program.

An added benefit of the use of multiple queues we propose is the organizational aspect it provides. This means, for instance, if a testing input results in a crash for a specific program, it becomes simple to identify the other variants that might also suffer from the same problems.

Almost all other modules below interact with the Queue Handler, be it for simple queue addition when a test case is deemed interesting or for obtaining an element of the queue for the splicing mutation phase (explained next).

2) Next Test Case Selection: Chooses the next test case to fuzz and mutate. The selection of the next test input to fuzz is

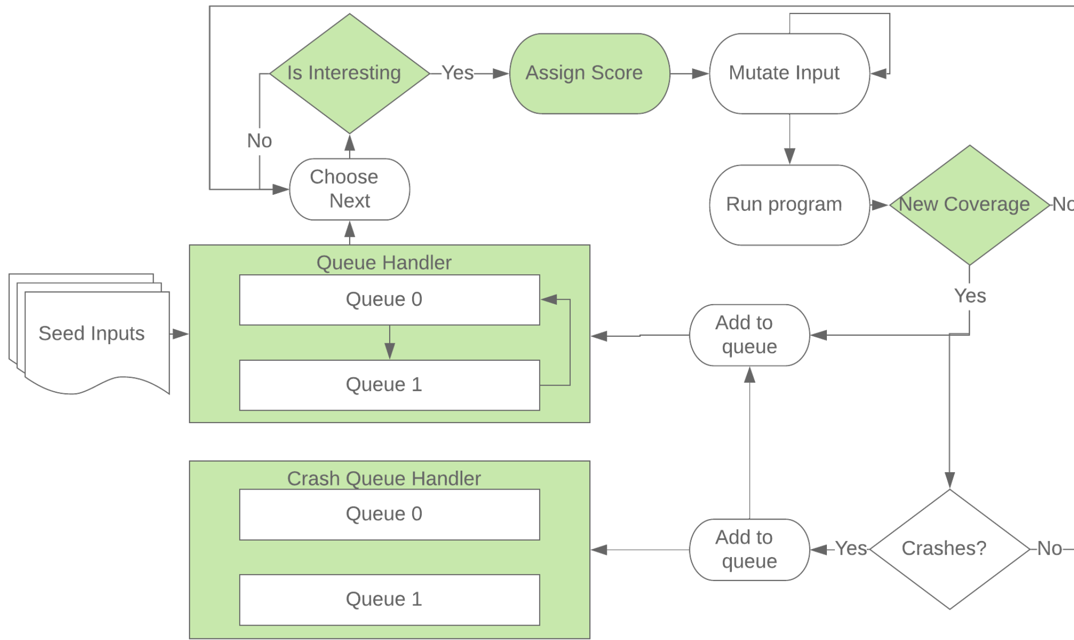

Fig. 3. Proposed fuzzing procedure architecture data flow.

a highly important subject that can make or break a fuzzer. If the fuzzer followed a random selection criteria, it could waste time with either tests that have already been performed or with tests that are less likely to trigger new coverage. A test case is considered as more or less likely to trigger new coverage based on the score assigned to it. Therefore, this module determines which test case should, or should not, be fuzzed next. As such, it interacts with all the modules that have to process the test case chosen, such as the mutation module. To simplify, we propose to evaluate and mark each test case as interesting or not interesting; when selecting a test case, the module simply picks the interesting test cases for fuzzing and skips the others.

3) Is Interesting: Unlike AFL, where an interesting test case is deemed interesting simply if it uncovered new edges/basic blocks in the program, our approach considers a test case interesting not only if it triggers new coverage in the SUT, but also if it uncovers new information in the variants of the program being tested. This is so we can prioritise inputs that trigger never before seen functionalities in any variant, avoiding repeating tests that lead to the same result.

To do so, all test cases, both initially given and generated during the fuzzing procedure, are deemed either interesting or uninteresting. By default, uninteresting test cases are only fuzzed after all interesting test cases have been processed.

4) Assign Score: Each test case has an associated score (or energy) that is used to determine how much effort the fuzzing process does to find and uncover vulnerabilities or paths in the program. The score takes into consideration the number of new paths uncovered, how far along the fuzzing process the test case was found, the size of the test case and how long the program takes to process the test case is. This score is calculated as the same way as AFL.

As mentioned before, one of the goals of our solution is to avoid repeating work that has already been done in previous fuzzing runs. This goal can be stated as focusing the fuzzing efforts on any previously unseen functionality or code block. That is, any path of code blocks that is tested on the exact same circumstances only in different programs will normally not reveal any further information from what it would reveal from one of the other variants. Taking this into consideration, all fuzzing efforts to understand more about the structure of a single program can be applied on the structure of another variant, as long as the two share some sort of functionality or code. Using this information, each test case has a score directly related to its performance in the specific program it was being executed on. The larger the score a test case has, the more mutation time is given, allowing for more inputs to be generated and developed from the said test case.

5) Mutate Input: The mutation process is exactly like the one found in AFL. It tries to find interesting test cases by both deterministic mutations, done once for each test case, and havoc mutations, which are random mutations of the test case itself. The number of mutations done to the test case depends heavily on the associated score.

6) Crash Queue Handler: The approach comprises a crash queue for every single variant. This is done mostly so the data storage is more organized, and the user has a simpler way to identify which test case inputs crash which variant.

VI. PandoraFuzzer Implementation

We implemented our greybox fuzzer approach in a fuzzer called `PandoraFuzzer`.

As we stated before, our tool is based on AFL and detects the same types of vulnerabilities as AFL. Our fuzzer was implemented mostly in C but has a single component of the

instrumentalization phase that was implemented in assembly code. This is the code that is injected in the binary of the SUT. This component is inserted by *gcc* or *clang*, allowing the collecting of coverage information and supporting the forkserver module (see Section IV-A5).

We applied a set of code modifications to AFL and added new nodules to it. To better describe the differences between the two fuzzers, we describe in the next section what was changed from the original fuzzer and why, explaining the logic behind those modifications and discussing some alternatives that were considered at the time. This is presented alongside the reasoning behind the decision-making process when choosing between each of the alternatives. Fig. 3 highlights in green the modules that were modified in the original AFL architecture to create our own fuzzer solution.

A. Main AFL Modifications

1) Program Instrumentalization: One big difference that allows for better internal program structure understanding, is the way we instrumentalize the programs to be tested. The edge identifier is no longer randomly generated, like in AFL, but it is based on the contents of the basic block itself. This allows the comparison of coverage information among program variants, which solves the problem number one in Section III-A. In `PandoraFuzzer` figuring out if two program variants contain the same functionality, or basic block, consists of simply checking if the programs share a basic block identifier. In our solution, the identifier corresponds to a summary of the contents of each basic block, which is represented by a hash, as explained in Section IV-B.

2) Multiple Forkserver Usage: This AFL modification will address the second challenge presented in Section III-A. Unlike AFL, where there is a single forkserver for a program, we have multiple forkservers for the various variants. This is done because each forkserver can only interact with the assembly code of a single program, making it impossible to have one forkserver for multiple programs. Hence the only logical choice was to have multiple forkservers, so we can have each one of them interact with their own respective program. As a side effect, we also simplify the way we obtain the coverage information since we do it much like AFL but maintain a variable that tells us from which variant the information is coming from.

3) Multiple Program Transition and Usage: By definition, single program vulnerability detection tends to lack the motivation to explore other programs besides the SUT. This creates the necessity to give to the fuzzing solution a mechanism where it can explore and discover interesting testing inputs for each and all of the variants. The solution to this issue resides on simple program switching when a given time interval has passed, allowing each variant to be fuzzed a similar amount of time, and every so often share what was learned with past tests. However, to get this solution more than one solution was experimented. For instance, multiple procedures, each running and fuzzing a single program was tried, since this allowed for each program to be fuzzed at the same time, theoretically allowing for faster results. However, the execution speed of each program left a lot to be desired because the more processes we have, the slower each process will be. This coupled with the fact that the solution would require an amount of cores that would have a linear growth as the number of programs become larger. Hence, it was decided to employ a single process switching among program variants while doing the fuzzing.

4) Program Input Organization using Multiple Queues: Since AFL focuses on fuzzing a single program, a lone test case queue is enough to store both the results of previous fuzzing operations, but also to organize the internal logic of the program. Unlike AFL, however, our approach might fuzz more than a program. Hence, there is a need for multiple queue management in the fuzzing mechanism.

5) Interesting Program Code Block Identifier Retrieval: In order to be able to identify which basic blocks the current input triggered, our fuzzer needs to have a way to be able to track code coverage in any given program. As such, our solution resorts much to the same approach of AFL, with one key difference. The approach of AFL consists of writing the edge to shared memory every time the execution passes through a basic block. The key difference consists on the way the basic block identifiers are generated before they are written into shared memory. We use edge coverage instead of simply tracking each basic block, allowing the collection of more structural information about the path being tested. To do so, we propose to try to directly target specific areas in the program variants by first identifying those areas and then prioritising inputs that passed through those areas.

6) Interesting Program Input Sharing: Simply fuzzing all program variants would yield no more interesting results than to simply fuzz those variants independently, one at a time, for the same amount of time. Furthermore, to avoid repeating tests that have already been done and would bring no different result, our approach proposes a mechanism that allows the fuzzer to learn from all the previous tests that have already been done. A mechanism which will allow it both to avoid repeating tests and also to help it uncover new paths, which are based on the paths already uncovered while fuzzing one of the variants.

The mechanism derives from the following concept. Whenever the fuzzer is switching between programs to fuzz, all the previously interesting inputs uncovered for the earlier variant that was fuzzed are executed on the next program. If the fuzzer finds any new coverage, i.e., any basic block that was not identified before, the fuzzer will designate it as interesting and save it for later tests. Then, the program input is marked as already checked for that program with the goal to avoid running the input more than once. In case the input is not considered interesting, it is simply marked as not useful for the specific program variant.

This mechanism avoids repeating work because all the previous tests that have been done are copied into the new testing program queue, excluding all the mutations and exploration that derived those inputs. At the same time, it allows the

tool to learn all paths and crashes that are uncovered by the previous tests. This proposed mechanism will address the last issue stated in previous section.

VII. Experimental Evaluation

Our experimental tests compare the results of the `PandoraFuzzer` with AFL, demonstrating the capability to detect potential bugs and the ability to avoid repeating work between program variants.

That way, the goal of the experimental evaluation was to answer the following questions: (1) Is the tool capable of detecting all vulnerabilities AFL is able to? (2) Is the tool able to learn from a previously tested program? (3) Is the tool at least as efficient as AFL at detecting vulnerabilities?

A. Testing Setup

The testing was performed on four different applications from two versions of *binutils*, a package containing several utilities distributed with the *Linux OS*. In order to test whether an application could learn from a previously executed testing procedure, two versions of each application were used. We first began by executing `PandoraFuzzer` for a single hour, fuzzing each application along with their different version. Afterwards, each application was fuzzed for 8 hours in the same conditions. Finally, the results were compared with AFL to be able to identify if we could find all vulnerabilities and to determine if we are at least as effective as AFL. To obtain the most precise values possible, metrics were used from the work of G. Klees et al. [26]. As such each program version was run for a total amount of ten times, the results presented here represent the average of those ten runs and the initial input was the same for each program.

The evaluation was conducted in the following steps: (1) instrumentalizing the four binutils programs and their versions using the process described in the Section IV; (2) Instrumentalizing a copy of the programs and their variants with AFL instrumentalization; (3) running the `PandoraFuzzer` tool for 1 hour and then for 8 hours with a given program and their variant program; (4) running a given program for 1 hour and then for 8 hours using AFL; then, doing the same for the variant of the program; (5) Compare the results when AFL is run individually in both variants and when using our tool for both programs at the same time.

B. Applications under test

Table I provides relevant important information about the binutils applications under test, such as the total number of files per program and the number of paths found during the evaluation procedure with either AFL or `PandoraFuzzer`. The applications are *cxxfilt*, *readelf*, *strings* and *size*, and the two versions of binutils used are V2.25 and V2.26.

C. Vulnerability Detection

This section contains an evaluation on the efficiency of `PandoraFuzzer` at discovering vulnerabilities on the binutils applications and compares it with AFL. We report the total number of unique crashes detected for both versions of binutils as they provide an indication of potential vulnerabilities (bugs) that are triggered with particular test cases. Table II shows the results for both tools for a testing period of 1 hour and 8 hours.

TABLE I
binutils APPLICATIONS CHARACTERIZATION.

Version	Program	LoC	Total Files	Total paths
V2.25	cxxfilt	5994	17	2739
	readelf	13275	3	644
	strings	5899	16	81
	size	5807	14	758
V2.26	cxxfilt	5816	17	2703
	readelf	14315	3	281
	strings	5895	16	80
	size	5799	14	694

TABLE II
UNIQUE CRASHES OBSERVED IN BINUTILS APPLICATIONS.

		Testing 1 hour		Testing 8 hours	
Version	Program	AFL	PandoraFuzzer	AFL	PandoraFuzzer
V2.25	cxxfilt	100	123	406	505
	readelf	0	0	0	0
	strings	0	0	0	0
	size	12	8	18	12
V2.26	cxxfilt	95	140	497	561
	readelf	0	0	0	0
	strings	0	0	0	0
	size	0	0	0	0

Fig. 4 depicts the average number of crashing test cases detected by `PandoraFuzzer` and AFL in Binutils V2.25 for one-hour tests and for eight hours tests. As the figure indicates, our tool was able to identify more crashing inputs than AFL.

The results answer both the first and last questions. The first question is answered since all types of vulnerabilities AFL found here were also found by the tool. The last question is answered since `PandoraFuzzer` managed to, on average, detect more unique crashes than AFL.

D. Code Coverage

To understand the difference between the code coverage achieved while fuzzing two programs, we used the number of paths found during the execution of a given program. Table III shows the total number of paths discovered by both tools for the four applications. While we can see an increase in the number of paths found by `PandoraFuzzer` in binutils V2.25 that does not occur in binutils V2.26. This is due to the fact that `PandoraFuzzer` focused most of the test cases generated on the code that was changed between versions, thus avoiding repeating the work previously done.

On Fig. 5 we can see how the tool compares against AFL when finding new coverage information on binutils V2.25 for one-hour tests, while Fig. 6 demonstrates the same for binutils V2.26. The results present in Fig. 7 and Fig. 8 show the exact same thing for eight-hour tests. All in all, it is possible to observe that our tool is able to increase code coverage when compared with the coverage provided by AFL.

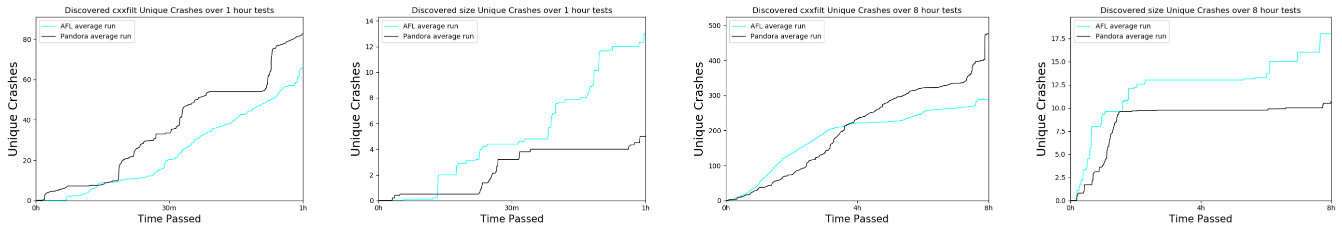

Fig. 4. Detections over a period of 1 hour (first two graphics) and 8 hours (last two graphics) for fuzzing tests

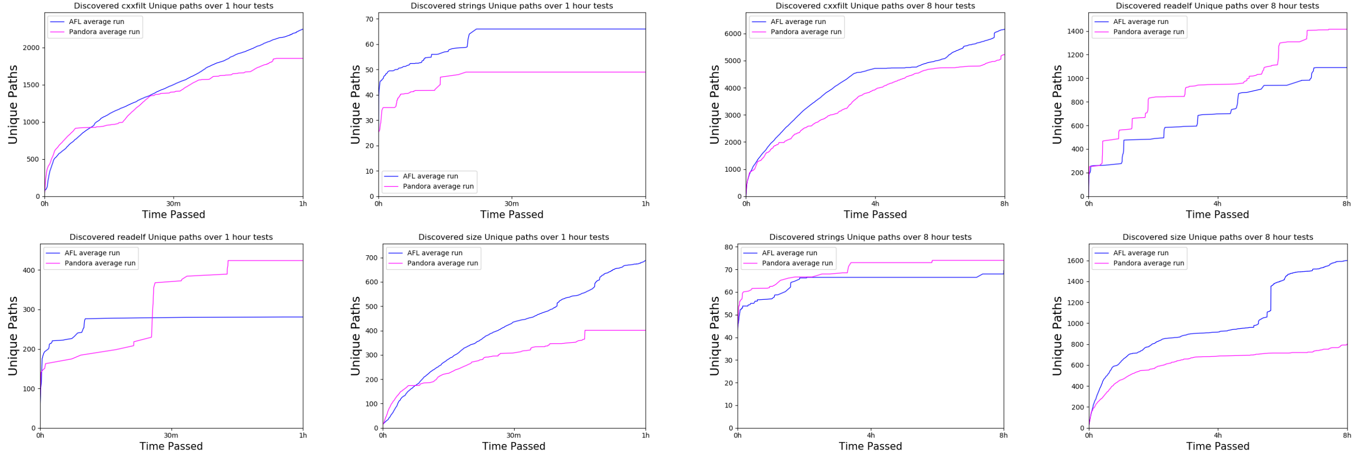

Fig. 5. Coverage 1 hour testing for binutils V2.25.

Fig. 7. Coverage 8 hour testing for binutils V2.25.

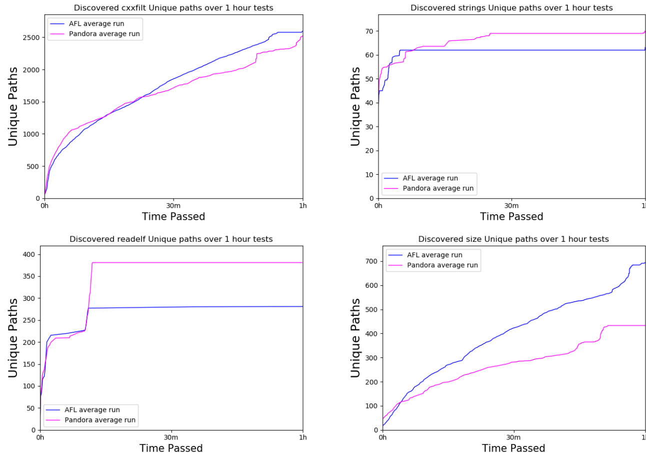

Fig. 6. Coverage 1 hour testing for binutils V2.26

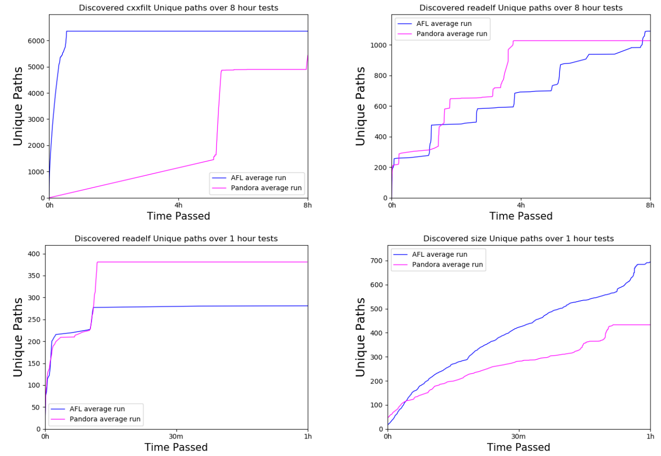

Fig. 8. Coverage 8 hour testing for binutils V2.26.

TABLE III
MAXIMUM NUMBER OF PATHS FOUND IN BINUTILS

Version	Program	AFL	PandoraFuzzer
V2.25	cxxfilt	5811	5962
	readelf	1090	1415
	strings	72	74
	size	1599	1143
V2.26	cxxfilt	6597	5820
	readelf	1090	1028
	strings	75	71
	size	1443	1057

To collect coverage information, we resorted to the afl-cov [27]. Afl-cov uses the test case files produced during the fuzzing phase to generate code coverage results for each program being tested.

Each binutils version had its four programs run once with afl-cov. Table IV shows the obtained results. The results were manually analysed, and they indicate that `PandoraFuzzer` focuses, around 20% more, on the changes that appear in the code of in the program variants, with the areas that did not change being mostly tested by the interesting inputs found in the previous fuzzing sessions. This answers the second

question, since the focus of the fuzzing efforts changed when subjected to previously learned information.

TABLE IV
RESULTS FROM AFL-COV APPLICATION UNDER THE FOUR PROGRAMS.

		Testing 1 hour		Testing 8 hours	
Version	Program	AFL	PandoraFuzzer	AFL	PandoraFuzzer
V2.25	cxxfilt	2174	1746	6157	5086
	readelf	280	412	1062	1369
	strings	66	47	74	68
	size	694	384	1555	787
V2.26	cxxfilt	2503	2482	6073	5122
	readelf	279	374	1068	1022
	strings	71	60	76	69
	size	667	397	1589	688

Overall, on average over all four binutils applications, `PandoraFuzzer` was able to detect more unique crashes and a higher percentage of the total paths in a given amount of time when compared with AFL.

VIII. CONCLUSION

The paper presents a greybox fuzzing approach and the `PandoraFuzzer` tool for automatic vulnerability detection in product variants written in the C and C++, utilizing the results of previous testing sessions of program variants to further boost code coverage and the number of vulnerabilities detected. The approach works as a mechanism able to identify shared functionalities among variants, alongside with the capability to learn from previous fuzzing sessions, and finally introducing the ability to avoid repeating tests that would only trigger functionalities that had already been tested to a certain degree of confidence in a previous fuzzing session. The tool developed was built upon AFL and utilizes an instrumentalization mechanism that differs from AFL to be able to identify shared functionalities between program variants. The tool also utilizes a learning mechanism so that every test case that has been previously generated while testing a variation of the program will no longer be executed. The experimental results showed that the tool is able to detect more unique crashes and a higher percentage of total paths in a given amount of time when compared with AFL.

ACKNOWLEDGMENT

This work was partially supported by the ITEA3 European through the XIVT project (I3C4-17039/FEDER-039238), and national funds through FCT with reference to SEAL project (PTDC/CCI-INF/29058/2017, LISBOA-01-0145-FEDER-029058, POCI-01-0145-FEDER-029058), and LASIGE Research Unit (UIDB/50021/2020).

REFERENCES

[1] CVE, “CVE Details. The ultimate security datasource,” https://www.cvedetails.com/browse-by-date.php.
[2] WhiteHat Security, “Application Security Statistics Report. The case for DevSecOps,” Nov. 2017.
[3] M. Zawlewski, “American Fuzzy Lop (AFL) Fuzzer,” http://lcamtuf.coredump.cx/afl/. 2017, [Accessed in 01/03/19].
[4] ——, “AFL Technical Details,” http://lcamtuf.coredump.cx/afl/technical_details.txt.
[5] “Common Weakness Enumeration,” https://cwe.mitre.org/data/index.htm/.
[6] X. Jia, C. Zhang, P. Su, Y. Yang, H. Huang, and D. Feng, “Towards Efficient Heap Overflow Discovery,” in *Proceedings of the USENIX Security Symposium (USENIX Security 17)*, Aug 2017, pp. 989–1006.
[7] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, “SOK: (State of) The Art of War: Offensive Techniques in Binary Analysis,” in *Proceedings of the IEEE Symposium on Security and Privacy*, May 2016, pp. 138–157.
[8] J. Cai, P. Zou, J. Ma, and J. He, “A Taint Based Smart fuzzing Approach for Integer Overflow Vulnerability Detection,” in *Proceedings of the IEEE International Conference on Software Testing, Verification and Validation*, 2014.
[9] I. Medeiros and M. Correia, “Finding vulnerabilities in software ported from 32 to 64-bit CPUs,” in *Proceedings of the International Conference on Dependable Systems and Networks (DSN), (fast abstract)*, 2009.
[10] N. Nethercote and J. Seward, “Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation,” in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Jun 2007, pp. 89–100.
[11] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, “LAVA: Large-Scale Automated Vulnerability Addition,” in *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, May 2016, pp. 110–121.
[12] M. Muench, J. Stijohann, F. Karglz, A. Francillon, and D. Balzarotti, “What you corrupt is not what you crash: Challenges in fuzzing embedded devices,” in *Proceedings of the Network and Distributed Systems Security Symposium*, Feb, pp. 18–21.
[13] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos, “Memory Errors: The Past, the Present, and the Future,” in *Proceedings of the Research in Attacks, Intrusions, and Defenses*, D. Balzarotti, S. J. Stolfo, and M. Cova, Eds., September 2012, pp. 86–106.
[14] A. Takanen, “Fuzzing for Software Security Testing and Quality Assurance,” Tech. Rep., June 2008.
[15] P. Oehlert, “Violating assumptions with fuzzing,” in *IEEE Security and Privacy*, vol. 3, Number 2, no. 2, March 2005, pp. 58–62.
[16] M. Woo, S. K. Cha, S. Gottlieb, and D. Brumley, “Scheduling Black-box Mutational Fuzzing,” in *Proceedings of the ACM SIGSAC Conference on Computer, Communications Security*, Nov 2013, pp. 511–522.
[17] P. Godefroid, M. Y. Levin, and D. Molnar, “SAGE: Whitebox Fuzzing for Security Testing,” *Queue*, vol. 10, no. 1, pp. 20–27, jan 2012.
[18] E. Bounimova, P. Godefroid, and D. Molnar, “Billions and billions of constraints: Whitebox fuzz testing in production,” in *Proceedings of the International Conference on Software Engineering (ICSE)*, May 2013, pp. 122–131.
[19] H. Chen, Y. Xue, Y. Li, B. Chen, X. Xie, X. Wu, and Y. Liu, “Hawkeye: Towards a Desired Directed Grey-box Fuzzer,” in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, Oct 2018, pp. 2095–2108.
[20] I. Haller, A. Slowinska, M. Neugschwandtner, and H. Bos, “Dowsing for Overflows: A Guided Fuzzer to Find Buffer Boundary Violations,” in *Proceedings of the USENIX Security Symposium (USENIX Security 13)*, Aug 2013, pp. 49–64.
[21] S. Karamcheti, G. Mann, and D. Rosenberg, “Adaptive Grey-Box Fuzz-Testing with Thompson Sampling,” in *Proceedings of the ACM Workshop on Artificial Intelligence and Security*, Oct 2018, pp. 37–47.
[22] “libfuzzer,” https://llvm.org/docs/LibFuzzer.html, 2018.
[23] “honggfuzz,” https://github.com/google/honggfuzz/. 2018, [Accessed in 21/02/19].
[24] A. Rebert, S. K. Cha, T. Avgerinos, J. Foote, D. Warren, G. Grieco, and D. Brumley, “Optimizing Seed Selection for Fuzzing,” in *In Proceedings of the USENIX Security Symposium*, Aug 2014, pp. 861–875.
[25] G. Grieco, G. L. Grinblat, L. Uzal, S. Rawat, J. Feist, and L. Mounier, “Toward Large-Scale Vulnerability Discovery Using Machine Learning,” in *Proceedings of the ACM Conference on Data and Application Security and Privacy*, ser. CODASPY ’16, Mar 2016, pp. 85–96.
[26] G. Klees, A. Ruef, B. Cooper, S. Wei, and M. Hicks, “Evaluating Fuzz Testing,” in *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, Oct 2018, pp. 2123–2138.
[27] “afl-cov,” 2018, https://github.com/mrash/afl-cov.