UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Generating Software Tests to Check for Flaws and Functionalities

Francisco João Guimarães Coimbra de Almeida Araújo

Mestrado em Engenharia Informática
Especialização em Engenharia de Software

Dissertação orientada por:
Prof. Doutor Nuno Fuentecilla Maia Ferreira Neves
e co-orientado pela Prof. Doutor Ibéria Vitória de Sousa Medeiros

2019

Acknowledgments

First of all, I would like to express my thanks to my advisors, Prof. Nuno Neves and Prof. Ibéria Medeiros for their guidance. Without their orientation and feedback, this dissertation would have never been completed in time.

I would also like to thank, by order of most distracting to least distracting, Nuno Burney, Robin Vassantlal, João Becho, Ana Maria Fidalgo, João Pinto, João Batista, Nuno Rodrigues, Tiago Correia, and Ana Ilhéu. Without all of you, I would have surely finished my thesis much sooner.

Lastly, I would like to express my gratitude to my family and especially my two dogs, Godzilla, and Thor for all the good distractions and helpful tips on how to best write a dissertation.

This work was partially supported by the PT2020 through the project XIVT (PORL/39238/2018), and by the FCT through the project SEAL(PTDC/CCI-INF/29058/2017), and LASIGE Research Unit (UID/CEC/00408/2019).

In memory of my Grandparents.

Resumo

O rápido crescimento da complexidade de software unido com a grande necessidade de software no dia a dia causou uma exigência para testar os mesmos de modo a conseguir garantir um certo nível de qualidade, funcionamento e segurança. Por exemplo, tanto o carro que conduzimos hoje como o frigorifico que usamos para manter a temperatura desejada dos nossos alimentos, requer software de tal complexidade que quando postos sobre alto stress, poderiam apresentar algum tipo de bug. No caso desse bug ser uma vulnerabilidade, e, por conseguinte, poder ser explorada, seria capaz de por vidas em perigo e mesmo causar danos financeiros no valor de milhões de euros. Essa vulnerabilidade conseguiria, por exemplo, criar a hipótese ao atacante de tomar controlo do carro ou, no caso do frigorifico, aumentar a temperatura fazendo com que a comida se estrague. Não obstante a isso, depois de essas vulnerabilidades terem sido descobertas, é necessário iniciar um processo de correção do software, custando tempo e dinheiro.

A complexidade do software cresce quando é necessário criar variantes das aplicações a partir de diversos componentes de software, como acontece em sistemas embebidos. Tal complexidade dificulta o teste e a validação do software para as funcionalidades que foi desenhado, podendo aumentar também o número vulnerabilidades de segurança. Estas vulnerabilidades podem permanecer ocultas durante vários anos em qualquer programa, independentemente de quantos testes foram executados para tentar assegurar a sua qualidade e segurança. Isto é tanto devido à eficiência destes testes que podem ser de uma qualidade limitada, bem como ao curto tempo disponível para garantir a correta funcionalidade. Um atacante externo, ao contrário, possui tempo teoricamente ilimitado para explorar o software quando este já se encontra no mercado.

Vulnerabilidades são a principal causa de problemas de segurança e o foco principal quando os atacantes estão a tentar explorar o sistema. Estes, podem também causar diversos tipos de danos ao sistema e aos stockholders da aplicação, como por exemplo o dono da aplicação e os utilizadores. Uma distinção importante é que nem todos os bugs são vulnerabilidades. Uma vulnerabilidade tem de ser explorada de modo a possibilitar a corrupção do comportamento normal do programa, levando a um estado erróneo deste.

De modo a conseguir tomar partido de um pedaço de software, os atacantes externos necessitam apenas de conseguir encontrar uma vulnerabilidade. No entanto, os testes desenvolvidos pelos responsáveis pela qualidade de segurança têm de encontrar inúmeros. Como resultado disto, hoje em dia as companhias gastam recursos em termos de custo e de tempo para conseguirem melhorar o processo de verificação e validação de software, por forma a tentar garantir o nível de qualidade e segurança desejado em qualquer dos seus produtos. No entanto, como acima referido, os recursos e tempo são limitados nos testes, fazendo com que vários bugs e vulnerabilidades possam não ser detetados por estes testes, mantendo-se ainda nos produtos finais. Embora já existam ferramentas automáticas de validação de segurança, não existe nenhuma ferramenta que possibilite a reutilização de resultados de testes entre versões de aplicações, de modo a validar estas versões e variantes da maneira mais eficiente possível.

Validação de Software é o processo de assegurar um certo nível de confiança, que o software corresponde às espectativas e necessidades do utilizador e funciona como é suposto, não tendo nenhuma incoerência de comportamento e tendo o menor número de bugs possível. Neste contexto, cada teste examina o comportamento do software em teste de modo a verificar todas as condições mencionadas anteriormente e contribui para aumentar a confiança no sistema em si. Normalmente, esta verificação é feita com conhecimento à priori do programa a ser testado. Isto, no entanto, é um processo muito lento e pode ser sujeito a erros humanos e suposições sobre o programa a ser testado, especialmente se forem efetuadas pela mesma pessoa que fez o programa em si.

Existem várias técnicas para testar software de maneira rápida, automática e eficiente, como por exemplo fuzzers. Fuzzing é uma técnica popular para encontrar bugs de software onde o sistema a ser testado é corrido com vários inputs semivalidos gerados pelo fuzzer, isto é, inputs certos o suficiente para correr no programa, mas que podem gerar erros. Enquanto o programa está a ser submetido a todos os testes, é monitorizado na espectativa de encontrar bugs que façam o programa crashar devido ao input dado. Inicialmente, os fuzzers não tinham em consideração o programa a ser testado, tratando-o como uma caixa preta, não tendo qualquer conhecimento sobre o seu código. Assim, o foco era apenas na geração rápida de inputs aleatórios e a monitorização desses inputs na execução do programa. No entanto, estes poderiam levar muito tempo para encontrar bugs somente atingíveis após certas condições logicas serem satisfeitas, as quais são pouco prováveis de ser ativadas com inputs aleatórios. A fim de resolver esse problema, um segundo tipo de fuzzers foi desenvolvido, whitebox fuzzers (fuzzers de caixa branca), que utilizam inputs de formato conhecido de modo a executar de maneira simbólica o programa a ser testado, guardando qualquer condição lógica que esteja no caminho de execução

de um input, para depois as resolver uma a uma e criar novos inputs a partir das soluções dessas condições. No entanto, a execução simbólica é bastante lenta e guardar as condições todas leva a uma explosão de condições a serem resolvidas perdendo muito tempo nelas. De modo a resolver estes problemas com o whitebox fuzzers (fuzzers de caixa branca), foram criados greybox fuzzers, uma mistura dos dois tipos de fuzzer descritos anteriormente que usa instrumentação de baixo peso para ter uma ideia da estrutura do programa sem necessitar analise previa causando muito tempo nessa instrumentalização, mas compensado com a cobertura devolvida.

No entanto, não existe nenhuma ferramenta, ou fuzzer, que consiga usufruir de informação obtida de testes realizados a versões mais antigas de um dado software para melhorar os resultados dos testes de uma versão do mesmo software mais recente. Hoje em dia, dois produtos que partilham funcionalidades implementadas de maneira semelhante ou mesmo igual irão ser testadas individualmente, repetindo assim todos os testes que já foram realizados no outro programa. Isto representa, claramente, uma falta de eficiência, perdendo tempo e dinheiro em repetições de testes, enquanto outras funcionalidades ainda não foram testadas, onde provavelmente podem existir vulnerabilidades que continuam por não ser descobertas.

Este trabalho propõe uma abordagem que permite testar variantes ainda não testadas a partir de resultados das que já foram avaliadas. A abordagem foi implementada na ferramenta PandoraFuzzer, a qual tem por base a aplicação de fuzzing American Fuzzy Lop (AFL), e foi validada com um conjunto de programas de diferentes versões. Os resultados experimentais mostraram que a ferramenta melhora os resultados do AFL.

A primeira etapa consiste na compreensão das várias vulnerabilidades comuns em programas desenvolvidos em C/C++ e os modos mais comuns de detetar e corrigir tais vulnerabilidades. A segunda etapa deste projeto é a implementação e validação da ferramenta. Esta ferramenta vai ser construída sobre um Fuzzer guiado por cobertura já existente, AFL, e segue um princípio semelhante. A terceira etapa deste projeto consiste na avaliação da ferramenta em si, usando várias medidas de comparação e foi validada com um conjunto de programas de diferentes versões. Os resultados experimentais mostraram que a ferramenta melhora os resultados do AFL.

Palavras Chave: Fuzzing, Detecção de Vulnerabilidades, Testes de Cobertura, Testes de Software, Segurança no Software

Abstract

Industrial products, like vehicles and trains, integrate embedded systems implementing diverse and complicated functionalities. Such functionalities are programmable by software containing a multitude of parameters necessary for their configuration, which have been increasing due to the market diversification and customer demand. However, the increasing functionality and complexity of such systems make the validation and testing of the software highly complex. The complexity inherent to software nowadays has a direct relationship with the rising number of vulnerabilities found in the software itself due to the increased attack surface. A vulnerability is defined as a weakness in the application that if exploitable can cause serious damages and great financial impact. Products with such variability need to be tested adequately, looking for security flaws to guarantee public safety and quality assurance of the application. While efficient automated testing systems already exist, such as fuzzing, no tool is able to use results of a previous testable programme to more efficiently test the next piece of software that shares certain functionalities. The objective of this dissertation is to implement such a tool that can ignore already covered functionalities that have been seen and tested before in a previously tested program and give more importance to block codes that have yet to been tested, detect security vulnerabilities and to avoid repeating work when it is not necessary, hence increasing the speed and the coverage in the new program. The approach was implemented in a tool based on the American Fuzzy Lop (AFL) fuzzing application and was validated with a set of programs of different versions. The experimental results showed that the tool can perform better than AFL.

Keywords: Fuzzing, Vulnerability detection, Coverage testing, Software testing, Software security

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The swift growth of software complexity coupled with the universal use and need of software in everyday life, has caused an equally significant need to test those pieces of software to assure certain quality standards and security both to the company that developed the product and the users that use it. As an example, the car we drive to work requires extremely complex software which, when put under stress, could eventually present some sort of bug. If any of the existing bugs happen to be exploitable, i.e., if they happen to be a vulnerability, it could give an opportunity to an external attacker to control the car, putting lives in danger and causing millions in damages. Not only that, but bug correction after the flaw has been found and explored would result in high costs to repair, in addition to all the previous described costs.

Such vulnerabilities can stay undetected for several years in the programs, regardless of the many tests campaigns, they are submitted to before release, because of the limited testing time and the efficiency of the tests themselves used to find an unknown number of bugs. Attackers, on the other hand, have a theoretically infinite amount of time to discover those vulnerabilities and only need to find one to take advantage of 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 to assure a certain level of quality on all their products. However, as previously described, since they have limited resources, many bugs and vulnerabilities still manage to find themselves in the final products.

A bug is different from a vulnerability in the sense that every 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 can be used to hurt the stakeholders of an application, such as the users or other entities that need the application.

Software testing is a very complex and time-consuming endeavour. Software engineering has demonstrated that test cases are far more effective when not performed by the original programmers, since they are bound to have preconceptions when creating tests that cover the implementation, such as considering certain inputs as insignificant. For human testers to be effective, implementation understanding is required, including boundary and corner cases. Acquiring this knowledge is expensive in both time and conceptual effort. However, by simply supplying the program with randomly generated input, a considerable part of the input state space can be explored without requiring any form of human interaction. Not only that but the random generation of such tests reduces bias and includes items that are so different from what any developer or tester could have imagined, that it may trigger parts of the attack surface of the program that could have easily been missed by the software testers or developers.

1.1 Motivation

The security of critical software has become more and more relevant in everyday life, becoming unavoidable as time goes on, such as in a car while commuting to work or while travelling to a distant country in a plane. In case a vulnerability is exploited in any of the previous examples, it could lead to highly expensive and dangerous results. For example, the costs of updating the software in vehicles have reached billions annually and so, they could be significantly reduced if appropriated software tests were performed. Even if the individual cost could be made negligible if the number of products affected by the vulnerability was big enough, the costs would still be very high.

Unfortunately, most software usually suffers from being highly complex and diverse, employing many people to develop it and such complexities usually result in a considerable number of bugs. From an original piece of software, there are usually several code blocks that can be reused, normally in distinct modules. Bugs found in those modules and code blocks will be carried to any and all software products that happen to utilize them.

Nowadays, fuzzing is probably the most effective state-of-the-art vulnerability detecting approach. Fuzzing 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 the pieces of software under test (SUT). As such, a lot of redundant tests occur, which decreases the efficiency of the procedure. Not only that, but the tests provided by the fuzzers will not find many vulnerabilities as well as new

vulnerabilities because they are busy exploring already explored program paths, finding the same vulnerabilities over and over, with no advantage to the company conducting the testing.

This thesis focuses on the development of a greybox fuzzer, which is constructed by modifying the AFL fuzzer. The resulting fuzzer can detect software vulnerabilities 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 shared among programs. This is achieved by retaining the results of previous tests to keep track of the code blocks already tested. As such, we can minimize the amount of repeated test cases done and maximize the coverage on the unshared functionalities, saving precious time and money and providing vulnerability detection and quality assurance to the company conducting the tests.

1.2 Objectives

This thesis main objective is to create a tool that is able to detect vulnerabilities in various software written in C or C++, which share an unknown number of features, in an effective and efficient way. We propose both an architecture for the tool, presenting every module that constitutes it, and the development of a prototype as proof that it works. In order to develop such a tool, the main objective can be broken down into three sub-objectives.

The first objective focuses on getting an understanding of the various vulnerabilities found in software nowadays, as well as the state-of-the-art techniques that are used with the aim of detecting those vulnerabilities in real-world software.

The second objective naturally is related to the design and the implementation of such a tool, which facilitates and enhances vulnerability detection in certain software products that share an unknown amount of common functionalities.

The third objective focuses on the evaluation of the implemented tool. This is done by comparing the developed tool with existing tools, in order to collect more information about the performance behaviour and when compared to the state-of-the-art.

1.3 Contributions

The main contributions of the thesis are:

- A fuzzer architecture that can be used to detect vulnerabilities in multiple program variants. A program variant is defined as simply being a change in a single program, for example, the addition or removal of a few lines of code. The proposed architecture takes advantage of previously done test procedures

to best understand how to test a given program variant and be able to identify potential vulnerabilities.

- As a proof of concept a prototype that implements the architecture, called PandoraFuzzer, and shows: (1) the efficiency to detect crashing inputs (vulnerabilities) in a multitude of programs with a variety of proprieties and sizes; (2) the ability to replicate the crashes found during fuzzing by providing to the user the testing cases that crashed the programs; (3) the capacity to learn from previous fuzzing efforts as to more efficiently identify inputs that can crash the given programs.

- An experimental evaluation of the currently implemented prototype with four applications, to reveal the viability and effectiveness of the tool developed at finding crashing inputs.

The developed work supported the publication of a full paper (12 pages) at the INForum 2019, in the track of Security of Computational Systems: Francisco Araújo, Ibéria Medeiros and Nuno Neves, Geração de Testes de Software para Verificação de Faltas e Funcionalidades, at Simpósio Nacional de Informática (INForum), September 2019.

1.4 Thesis Structure

This thesis is structured as follows:

Chapter 2 explains some relevant concepts and provides fundamental context for the work done in this thesis. Distinct types of vulnerabilities are explored along with the different kinds of program coverage and code representation. Various types of fuzzers are also presented, like blackbox, whitebox and greybox fuzzers. Lastly, AFL, a greybox fuzzer, is described more in detail.

Chapter 3 is dedicated to explaining the proposed solution, including the architecture and main components. We also describe the way the components interact together to solve the problem addressed in the thesis. A few of the considered alternatives are also discussed.

Chapters 4 and 5 describe the current implementation of the proposed architecture in the PandoraFuzzer prototype as well as evaluating and validating the prototype with four applications. A detailed explanation is given about the testing platform the tests were conducted on. The results show the efficiency of the architecture proposed when compared with AFL.

Chapter 6 provides a conclusion of the developed research and discusses future work.

Chapter 2

Context and Related Work

This chapter explains the related works to this dissertation as well as some context that serves as the basis for the project. First, we describe the techniques employed in test generation to increase software code coverage. Secondly, we explain in detail most of the vulnerability classes we can find, describing how they occur and how they can be corrected and avoided. Afterwards, we present the state-of-the-art of the fuzzing mechanisms for bug discovery. Finally, we go into greater detail to describe AFL, a greybox coverage fuzzer where the tool will be built upon.

2.1 Software Validation and Verification

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 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. A test set, as the name indicates, is a set of test cases. An important distinction must be made between a fault, an error and a failure. A software fault is a defect in the software. 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.

2.1.1 Types of Coverage

Program coverage of a suite of test cases can be divided into: *Basic Coverage Criteria* and *Coverage Based on Graphs*. Those different criteria allow the development of tests suits that can result in better coverage of the SUT as well as giving a coverage goal to aspire to.

Basic Coverage Criteria

Three main coverages are relevant with regard to this criteria:

- *Line Coverage*: As the name indicates, it is the percentage of all the lines the programs that are executed by the test set. While it might be possible to provide 100% Line Coverage, it does not tell us much about the behaviour of the software.
- *Branch Coverage*: Percentage of branches, such as if statements or loops, that have been executed at least once while running the test suite. This criteria provides some better knowledge about the program but probably still not enough to get a good understanding of the SUT.
- *Instruction Coverage*: Percentage of instructions in the software that are executed by the test suite. An instruction is any line in the program and each component in a conditional branch.

As it might be evident, some of these criteria subsume the other, i.e., instruction coverage subsumes line coverage because for every test set that might satisfy the requirements of instruction coverage, also satisfies the requirements of line coverage, while the inverse is not true. One important feature missing from these coverage types is the order in which those conditions happen. The order might affect the state of the program at the time each line is executed, or a branch, or as atomic statement. The next coverage type remedies this limitation somewhat.

Graph Coverage

A graph can be used as the representation model of the SUT, where each node is a basic block. A basic block is defined as a sequence of code lines that are always executed together in a sequence, each edge represents the control flow between basic blocks. The execution of a test corresponds to a path in the graph, i.e., a set of basic blocks connected by edges. The requirements of this criteria is a test set that can cover the paths of the graph. Unlike the basic coverage criteria, in some graph coverage criteria the order of execution matters, making it a much better criteria to pick from. Such criteria includes:

- *Node Coverage*: Much like line coverage described before, Node Coverage is the percentage of nodes executed after running the test suite. This does not take into consideration the order on which the nodes are reached.
- *Edge Coverage*: Corresponds to the percentage of Edge executed after running the test suite, which corresponds to Branch Coverage since each edge represents a conditional statement.
- *Edge-pair Coverage*: It is the requirement to cover all paths up to length 2 in the graph. In other words, the requirement to execute each pair of nodes that is connected through an edge in the graph.
- *Prime Path Coverage*: A prime path is a maximal length simple path, i.e., a path that has no repeated nodes with the exception of possibly the first and the last, that is not a subpath of any other path. It requires every prime path to be executed by the test set. This can result in a vast amount of paths making it infeasible sometimes to test on large program where there are millions of paths.

In order to test a program more adequately, we could focus on the flows of the data values in the graph itself, while ensuring that the created values are always in a correct state and used appropriately. A variable can either be defined, in which case one assigns a value to the said variable or used, in which one accesses its value. So, given the previously defined graph, in each node we could declare which values were defined and used, allowing us to define that alternative criteria for code coverage. In the dissertation, we will, however, focus more on the criteria defined above.

Some paths, regardless of what coverage is being used, might end up being infeasible, meaning they might be impossible to cover. For example, a simple path that requires leaving a for loop the moment we enter it might be impossible depending on the for loop and the program under test.

2.1.2 Code Representation

While code can be represented in a simple graph, there are several types of graphs we can choose from. In particular, different representations of code have been developed to reason about the various properties of the software. Even if those representations have been designed to have a better understanding of the software and how to optimize it, they can also be used to generate software tests to form the basis of vulnerability detection. We present three different kinds of representations, namely abstract syntax trees, control flow graphs and program dependence graphs.

- *Abstract Syntax Trees* (AST): These trees are an intermediate representation produced by code parsers of compilers. They are ordered trees where inner

nodes are represented by, for example, additions or assignments. Inner nodes encode how statements are nested to produce programs and the leaf nodes are operands, such as constants or identifiers.

- *Control Flow Graphs* (CFG): A control flow graph describes the order in which code statements are executed along with the conditions that need to be met for a path to be taken. The nodes represent the statements, and the edges, which are directed, represent the transfer of control. This type of graphs helps guide fuzzing testing tools such as in the work done by S. Sparks et al. [31].

- *Program Dependence Graphs* (PDG): This type of graphs were originally developed to determine all statements and predicates of a program that affect the value of a variable. As previously mentioned, detecting the values of variables from where they were defined to where they are used is extremely important in software verification. The program graph uses two types of edges: data dependency edges and control dependency edges. Data dependency edges reflect the influence that one variable has on another. Control dependency edges correspond to very much the same thing but with predicaments on the variables.

Work has been done in the development of new code representations. F. Yamaguchi et al. [36] propose a new code representation which is a mixture of all the previous, called *code property graph* in order to better model templates for vulnerabilities in graph transversal. They are able to present small and concise traversals for many vulnerabilities like buffer overflows and memory address leaks. It is a directed graph where each property is assigned to edges and nodes, where graph traversals are the main way of obtaining information about the code. A traversal is a function that maps a set of nodes to another group of nodes. They combine the three representations in the following manner: they start by transforming the Abstract Syntax Tree into a program property graph, they do this afterwards to every single one of the previous code representations (AST, CFG, PDG) and then combine them. They evaluated the efficacy of their approach in coverage analysis and vulnerability detection by testing it on the source code of the Linux kernel. The evaluation showed that they obtained a better view of the code and could describe more vulnerabilities than any of the previous one individually and any combination of them.

2.2 Vulnerabilities

Vulnerabilities are the root cause of security problems and when they are exploited by attackers, they can cause damage to the system and the stakeholders, that is, anyone with interest in the system, such as users or the software owner. A distinction

must be made, a vulnerability is a bug but not all bugs represent a vulnerability. A vulnerability is a bug that can be exploited to corrupt the programs normal behaviour, leading it to an erroneous state. ISO 27005 defines a vulnerability as *"a weakness of an asset or group of assets that can be exploited by one or more threats"*. In this study, we consider the following vulnerabilities identified in the Common Weakness Enumeration (CWE) [1] as problems for the applications programmed in the C and C++ languages.

Variable Overflow and Underflow

Some of the most common security vulnerabilities in software are buffer overflow flaws. When this sort of bugs are exploited the running program may return corrupt information, crash or it might even allow the attacker to take control of the execution flow of the program. While this class of errors belong to Memory Errors, we decided to put them into their own specific section because of their importance. In this study, we focus on the following problems related to Buffer Overflows:

1. Buffer overflow and underflow: A buffer overflow occurs when a program tries to write data outside the memory allocated for the memory buffer. This can occur because there is a write before the start of the buffer (underflow) or after the end of the buffer (overflow).

2. Stack overflow and underflow: A stack overflow means that the bug occurs in a buffer stored in the stack.

3. Heap overflow and underflow: Similarly, it means a buffer stored in the heap has exceeded its allocated memory limit.

4. Global variable overflow and underflow: Occurs when one of the above happens in a buffer that is a global variable.

The following is a buffer overflow example, which occurs by copying the user input into the buffer without having any bound checking. If the size of the user input buff is bigger that the allocated value buffer, it will result in a buffer overflow. Function *strcpy* copies the contents of buff to buffer up to a *"\0"* character.

```
void buffer_overflow(char *buff){
  char buffer[5];
  strcpy(buffer, buff);
}
```

Any of these bugs can occur due to improper restrictions of operations in the arrays, or incorrect calculation on the index used in buffers, or even integer overflows.

In the study conducted by X. Jia et al. [19], the authors propose a new solution to discover potential vulnerabilities by modelling heap overflows as spatial inconsistencies between allocations and access to the heap. They implement a prototype called HOTracer which found a considerable number of previously unknown vulnerabilities using dynamic analysis, without employing test cases to directly trigger vulnerabilities. Initially, HOTracer pre-processes the sample inputs by selecting representative inputs, and then uses them in a dynamic analysis component to generate execution traces for each of them. For any given trace, HOTracer traverses it offline to do an in-depth analysis. It proceeds to identify any heap operation and builds the heap layout. HOTracer then tracks the heap objects spatial and taint attributes during the execution traces. Finally, it checks for any potential heap overflows and generates inputs to prove them.

The work of Y. Shoshitaishvili [30] focuses on vulnerability detection through a binary analysis tool, *angr*, that implements many existing approaches in offensive binary analysis. They designed *angr* with the intent to make it able to support cross-architecture. They do this since modern hardware architectures vary immensely and they require cross-platform support and to support all different types of operating systems. They also need the ability to support various analysis techniques, as well as the ability to be used easily by the software security community and most have the ability to reproduce the input that caused the vulnerability or replayability of the crash. The tool generates a CFG by performing an iterative CFG recovery. It starts from the entry point of the program, after the control Flow Graph is generated, they run a Value-set Analysis which is a static analysis technique that combines numeric analysis and pointer analysis for binary programs. In the evaluation phase, they show that after a crash is identified only a small percentage of the inputs were not easily identifiable.

Integer Vulnerabilities

In this study we also take into consideration integer overflows vulnerabilities. We take into consideration the following types char, short, int, long and long long, which can be used to store integers of different sizes. Any of this types, except char, are signed by default, while char is unsigned. If a value is signed, it means it can represent both negative and positive values.

Below we show a few examples of integer vulnerabilities:

1. Overflow
2. Underflow
3. Demotion/narrowing/truncation

4. Sign conversion

Overflows occur when the required number surpasses the known maximum limit of its representation. Independently of what type the overflow is, when it occurs it always has the same result, but not the same value, it loops back around. If we are at the maximum limit of a specific value and increment by one, we will have the minimum value as a result.

The following vulnerability occurs when the value passed to the function is equal to 101. It happens because, as explained before, the value returned passes the maximum representation value, looping the value around.

```
int integer_overflow(int a){
  if(a == 101){
    return INT_MAX + 1;
  }
  return a;
}
```

When an underflow occurs, much of the same happens, but instead of passing the maximum possible value that can be represented on the specific type, the progress passes the minimum possible value that can be represented which causes the value to loop around to the maximum value. As an example, if we subtract one from the minimum value of any given type, the value on that variable will be the maximum possible value in the represented type.

```
int integer_underflow(int a){
  return (a == 10002) ? INT_MIN - 1 : a;
}
```

Type narrowing occurs when one tries to save a value from a type that occupies more memory to a type that requires less, such as a 32 bit int to a 16 bit short. In this case, only the bottom 16 bits of the int are copied to the short. For unsigned numbers this might result in a loss of information if the value required more than 16 bits to be represented. If this happens in signed numbers, the truncation might change a negative value to a positive value or vice-versa.

As shown in the following example, we are trying to pass a value that requires all the allocated memory to a value that does not have that memory capacity. What occurs is that the unsigned int value gets shorten since it cannot be represented as an unsigned short.

```
void truncating_unsigned(int a){
  unsigned int val = INT_MAX;
  unsigned short ss = val;
}
```

Sign conversion happens when a signed integer is converted to an unsigned number or the other way around. If the most-significant-bit is a zero, then no issue will happen in the conversion. Otherwise, the change will result in a sign and value change as well.

The following example demonstrates a signed conversion that happens when we convert from an unsigned short to a signed one. The value 0x8080, which corresponds to 32896, will become in the variable *ss* the value -32640.

```
void sign_conversion(){
  unsigned short us = 0x8080;
  short ss = us;
}
```

Integer vulnerabilities are such a common cause of bugs that specific techniques have been developed to detect them in a variety of ways. In J. Cia et al. [10] a fuzzer was implemented called SwordFuzzer that uses taint analysis to identify which bytes in an input file might be relevant in security-sensitive operations. This allows the tool to only test those bytes and hence be more efficient for integer overflow vulnerability detection. The key idea is using taint analysis to determine which bytes are more relevant to be mutated. The evaluation concluded that the efficiency of the fuzzing is improved dramatically as drastically fewer bytes are required to be mutated when compared to other fuzzers.

Memory Vulnerabilities

Memory corruption bugs are one of the oldest and most important problems in computer security. Memory errors are especially common in low-level programming languages like C or C++. To put it simply, a memory error occurs when an object accessed using a pointer expression is different from the one intended. Even without considering buffer overflows, memory errors have been stuck in the top-3 of the CWE SANS top 25 most dangerous software error for a long time [28], and several attacks nowadays often start with a memory corruption that provides a crevice to start infection.

There are several types of memory errors, and a few examples are listed below:

1. Use after free: Using a pointer to a region of memory that has been freed previously.

2. Memory Leak: Occurs when dynamically acquired memory is not freed often it is no longer needed.

3. Double free: As the name indicates, occurs when a free operation is performed on an already freed pointer.

We can see from the following example on a memory leak. The function reserves some memory and then ends up forgetting to free it.

```
void memory_leak(){
  float *a = malloc( sizeof(float) * 45 );
  float b = 42;
  a = &b;
}
```

In M. Muench et al. [23] it is shown that memory vulnerabilities are a prevalent class and that simple liveness checks are insufficient to detect many kinds of bugs in a blackbox manner when dealing with embedded systems. This happens because memory corruptions can change behaviours depending on what type of system the vulnerability occurs on. For example, on embedded devices the results of a memory corruption are less visible than on desktop systems. Desktop systems have nowadays a plethora of mechanisms to detect faulty states, such as heap hardening and sanitizers, while embedded devices often do not have such mechanisms due to their limited I/O capabilities. As such, silent memory corruptions can occur more frequently when compared with traditional computer systems creating a bigger challenge for conducting fuzzing sessions. They also evaluate and describe heuristics that can be used at run time during the analysis of an embedded device in order to locate memory corruptions.

N. Nethercote et al. [24] describes a dynamic binary instrumentation framework called valgrind, that can detect memory vulnerabilities using shadow values. The technique shadows every register and memory value with a data structure that describes it. Valgrind has become a standard C and C++ development tool on Linux and Memcheck is the tool that looks for memory bugs. While valgrind does slow down the programs execution, it allows for better memory error detection while still having reasonable performance.

In V. van der Veen et al. [34] it is presented a study of memory error statistics analysing vulnerabilities and the occurrences of exploits in the past 15 years. It is shown for every memory error when it was first introduced along with the defensive measures that were implemented at the time and how they were surpassed. The paper claims that while the number of vulnerabilities has been decreasing, the number of exploits has not. It is said that the reason for the diminishing number of memory vulnerabilities starting in 2007 could be related to fundamental changes of web development and the fact that the software industry became more mature.

In the work done by B. Dolan-Gavitt et al. [13] it is described a tool, LAVA, which is able to introduce serious bugs into any program, such as a buffer overflow allowing the testing of tools that look for bugs. To add bugs to programs LAVA first identifies execution traces locations where input bytes are available that do not determine control flow and have yet to be modified, calling them DUA for *Dead, Uncomplicated and Available* data. From there, the tool finds potential attack points that are temporary after a DUA in the program trace and then it adds code to the program to mimic the vulnerability. In the evaluation phase, they first injected a large number of bugs into four open-source programs and then evaluated the distribution and realism of the generated bugs. Finally, they performed a preliminary investigation to see how effective an open-source fuzzer and a symbolic execution-based bug-finder were.

2.3 Fuzzing

Fuzzing is a popular technique for finding software bugs where the system under test is barraged with randomly generated test cases. It is used for security testing and quality assurance proposes, such as in the work done by A. Takanen [32], and by P. Oehlert [25], 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.

Most fuzzers differ in many significant ways as will be described further on, but, in general, they all follow a simple, yet highly effective algorithm, shown below.

```
General Fuzzing Algorithm{
  Queue <- Initial Test Cases of concrete valid input
  while ( not DoneWithFuzzing ){
    chosenTestCase <- chooseTestCase(queue)
    runProg(chosenTestCase)
        mutatedTestCase <- mutate(chosenTestCase)
        if (isInteresting(mutatedTestCase)){
          Queue <- addToQueue(mutatedTestCase)
        }
  }
}
```

This type of algorithm always receives, and returns, concrete valid inputs (or test cases) that the SUT (software under test) 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 software developers and testers that can use them to locate the bug and reproduce the crash.

In greater detail, the fuzzing tests can be implemented by three components:

1. Fuzz Generator: Responsible for generating the test cases that will be used to drive the SUT. The output is a series of inputs, which are subsequently fed to the SUT by the delivery mechanism.

2. Delivery Mechanism: Accepts system test cases from the fuzz generator and presents them to the SUT for consumption.

3. Monitoring System: Observes the SUT as it processes the input, attempting to detect any and every erroneous behaviour of the program, often translated into a program crash.

In the beginning, fuzzing tended to rely simply on the construction of the random test cases from a sequence of random numbers. However, fuzzers have evolved significantly over the years to the point where there are several classes of fuzzer. Classes of fuzzer can be categorized by how they implement the following three criteria:

1. Production of inputs/test cases

2. Before knowledge of input structure

3. Awareness of program structure

Regarding the generation of test cases, fuzzers can be divided into two categories, grammar based and mutational fuzzers. Grammar based fuzzers, described in the work done by P. Godefroid et al. [15], such as SPIKE and PEACH, construct inputs according to some user provided format specification, which imposes significant manual effort to create but allows for better coverage. It, however, fails to cover applications without known grammar. Mutational fuzzers, such as AFL, honggfuzz and zzuff, on the other hand, require no user effort to create any test cases since they "mutate" some initial program inputs. However, they usually require more time to have the same coverage as generational fuzzers.

Blackbox Fuzzers: These 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. However, even if they run extremely quickly, they can take an extremely long time to find deeply nested bugs due to the random nature of the input generation. In other words, blackbox fuzzing while still being extremely simple to the tester has the greatest ability to generate the largest amount of tests, but only provides limited coverage and so the testing can be very inefficient.

In the work done by M. Woo [35] it is developed an analytical framework using a mathematical model of blackbox mutational fuzzing. They model the repeated fuzzing of a configuration as a bug arrival process, modelling it as a weighted variant of the Coupon Collector's Problem, where each coupon type has its own fixed but initially unknown arrival probability. The Coupon Collector's Problem concerns a consumer who obtains one coupon with each purchase of a box of breakfast cereal. Suppose there are a variety of different coupon types in circulation, one basic question about the problem is what the expected number of purchases is required before the consumer amasses a number of unique coupons. In this variation, they model the coupons as the bugs and give them a weight which represents the chance of encountering those bugs, initially unknown. They develop FuzzSim, a replay-based fuzz simulation system, in order to model and evaluate online algorithms using pre-recorded data.

Whitebox Fuzzers: This type of fuzzers fix many of the faults 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.

As an example, the whitebox developed by P. Godefroid, SAGE [16], is a whitebox fuzzer for *Windows OS* applications, which, starting with a fixed input, symbolically executes the program gathering input constraints from conditional statements encountered along the way. The collected constraints are then negated and solved with a constraint solver, yielding new inputs that will go on to exercise different execution paths that were previously protected by those constraints. One key innovation behind SAGE is the algorithm called generational search for dynamic test generation, which is designed to partly explore the state spaces of large applications to avoid path explosion, since systematically executing all feasible program paths does not scale on large programs. The search algorithm also maximizes the number of new tests generated from each symbolic execution. The key innovation behind the algorithm, however, is the way the children test cases are used. Given a set of constrains collected during the execution of an input with each of them corresponding to a conditional statement. When the solution to the new path constraint is

used to update the old input, all test cases that were not used are preserved, and the ones that did are updated. From the experimental results, which were of limited size, they found several bugs that were missed by traditional blackbox fuzzers. They also noticed that symbolic execution is slower than testing or tracing a program.

In E. Bounimova et al. [8] it is shown the results of the whitebox fuzzer, SAGE [16]. In the paper, they describe the challenges with running the fuzzer in production as well as showing data on the performance of constraint solving and dynamic test generation. They claim that since 2017 SAGE has found hundreds of previously-unknown vulnerabilities. Notably, it was used to find many bugs in the development of Windows 7. They faced production challenges such as in a multi-week whitebox fuzzing that had to consume hundreds of gigabytes of disk, where each task in the SAGE pipeline increased the probability of something to go wrong.

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.

H. Chen et al. [11] describes a directed greybox fuzzer, Hawkeye, that combines static analysis and dynamic fuzzing. They developed this fuzzer with four main properties in mind. The directed greybox fuzzer (DGF) should have a well-defined and developed distance-based mechanism to guide the fuzzing while still considering all traces to the targets. This is done since there might still exist several traces towards the target that have yet to be explored. So, the guiding mechanism must find all the traces, or paths, that can lead to the target. Also, the DFG should strike a balance between overheads and utility of static analysis. The fuzzer should also schedule the testing inputs to reach the target site rapidly and it should adopt an adaptive mutation strategy when the test cases cover the different program states. When evaluating the tool, they strived to answer four questions: (1) check if static analysis beforehand is worth it, the experimental results shown that they outperformed the vanilla AFL most of the times and the cost of the static analysis was deemed worth it; (2) If Hawkeye could detect crashes more rapidly and more effective than any other tools; (3) are the dynamic strategies in Hawkeye effective; (4) how effective the tools capacity for reaching specific target sites.

Another study of the combination of static and dynamic fuzzing was done by I. Haller [18], in which the authors implement an evolutionary fuzzer called VUzzer. The fuzzer 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. In the beginning of the fuzzing process, VUzzer use lightweight static analysis

to compute the weights for each basic block of the application binary and then run the program on some inputs to determine the initial set of control-flow and data-flow features. In the evaluation phase, they compared it with AFLPIN [33], which has the same engine as AFL, and determined that VUzzer was able to find more bugs with a lot fewer inputs, concluding that inferring input properties by analysing application behaviour is a viable and scalable strategy to improve fuzzing performance.

S. Karamcheti et al. [20] 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. They argue that the best way to optimize fuzzing is to prioritize mutators operators that have been successful in the past. Then they demonstrate that tuning the mutation operator generates new sets of test cases that significantly improve the code coverage and also finds more crashes. In the evaluation, there was a comparison of Thompson Sampling against some fuzzers, such as AFL, for relative coverage in a testing period of 24 hours. They show that Thompson Sampling is extremely effective at the beginning of fuzzing and that there are significant gains to be made by improving existing fuzzing tools with data-driven machine learning techniques. They conclude that while its approach did not gain optimal results in all experiments, it worked in a vast majority of the real-world applications they tested it on.

LibFuzzer is a coverage-guided, evolutionary fuzzing engine to test C/C++ software [4]. LibFuzzer works by implementing a fuzz anchor. An anchor is a program written in C or C++ that allows the tester to specify a fuzzing entry point. This entry point is a function that accepts data and the size of the data. With this function, the tester can direct the fuzzer to whatever function it is desired, where it will then execute the fuzz target. This type of fuzzers require some sample inputs for the SUT. This corpus should optimally be packed with a variety of valid and invalid inputs. LibFuzzer generates random mutations based around the input given originally. If the fuzzer discovers new and interesting test cases, which is defined as any test case that covers new code paths, the test case is then saved for later usage or mutation. Another such fuzzer is honggfuzz [2], a security oriented, feedback-driven, evolutionary fuzzer. Honggfuzz has been used to find some interesting security problems in some major software packages [3].

Alexandre et al. [27] presented a way to optimize test case selection in order to increase coverage of the software under test. They show that current test input selection strategies found in Peach do not have better results than randomly picking the test cases. They also show ways to improve the test input selection strategies to maximize the total number of vulnerabilities found during fuzzing. Also, it can

be done in a manual or automated way while making no assumption of the type of fuzzer. This means it works for greybox fuzzers and whitebox fuzzers alike. The main objective of this study is to test different program input selection techniques and analyse them in order to conclude which one is best in what scenarios. The results of the experiments conducted the paper showed an increase in vulnerability detected when compared to Peach and a reduced testing set size. Allowing for more bugs to be found with fewer testing cases.

In the work by G. Grieco et al. [17], it is predicted if a test case is likely to discover software vulnerabilities by using lightweight static and dynamic features implemented using machine learning techniques. They do this mostly by analysing binary programs according to some procedure to perform the vulnerability discovery. They implemented a tool called VDiscover that uses two components: a fuzzer to mutate the original test case and a dynamic detection module to identify memory corruptions. Their proposed methodology works in two phases. A training phase, where they train the tool, and the recall phase where a trained classifier is used to predict if new test cases will find bugs or not, which can be later prioritized for further analysis. The results of the evaluation show that by analysing a small percentage of the test set pointed as potentially interesting, VDiscover can predict with reasonable accuracy which programs contain a vulnerability, which results in a significant increase in the fuzzing speed.

G. Klees et al. propose [21] some guidelines to better test and evaluate fuzzing algorithms. Of the 32 papers examined, they found that most experimental evaluations left a lot to be desired. They claim that for a new fuzzing algorithm it must be empirically demonstrated that it provides an advantage over another baseline fuzzer using a sample of target programs, that being the benchmark suite. An evaluation should also take into account the fundamentally random nature of fuzzing, since each fuzzer execution on the same program might result in different results. As such, an evaluation should measure sufficiently many trials to sample the overall distribution that represents the fuzzers performance, using a statistical test to determine if the new fuzzing algorithm is an improvement against the baseline, assuring that the improvement is real rather than being due to chance. When running the benchmark suite a performance metric is required to measure the fuzzing algorithm and suggest that reliance on heuristics for evaluating performance is not optimal. A better approach would be to measure against ground truth directly by assessing fuzzers against known bugs. After this, a meaningful set of program input files is required to start fuzzing with, and a timeout of a considerable long duration (>= 24 hours vs 5 hours).

2.3.1 Fuzzer Running Example

This section presents a simple example of how each basic fuzzer type tests a piece of code.

Consider the program shown below. The program takes as input 5 bytes and will trigger an error if the inputs are equivalent to *"bug!!"*. This bug is only executed if the value of the variable *count* is equal to 5 at the end of the function.

```
void bugFinder(char input[5]){
  int count=0;
  if(input[0] == 'b') count++;
  if(input[1] == 'u') count++;
  if(input[2] == 'g') count++;
  if(input[3] == '!') count++;
  if(input[4] == '!') count++;
  if (count == 5) abort(); //error
}
```

Blackbox Fuzzing

Any blackbox fuzzer would behave in a similar fashion. Starting with a random input, it would randomly generate hundreds of thousands of inputs until it triggered the bug. As it is understandable, running this program with random inputs hoping that the exact five-byte value is selected to trigger the bug is unlikely to be fast or efficient. There are 2^{8*5} possible input values that could have to be tested, with a probability of $1/2^{40}$ of actually triggering the bug.

Whitebox Fuzzing

Using as an example the previously talked whitebox fuzzer SAGE [16], we are going to demonstrate how it would look for the previously identified bug. Initially, SAGE would start with a random value perhaps given by the initial test case, lets say the initial input is *"good!"*. The symbolic execution would collect the following predicates: $input[0] \neq' b'$; $input[1] \neq' u'$; $input[2] \neq' g'$; $input[3] \neq'!'$; $input[4] =='!'$. To force the program through a different equivalence class, SAGE would compute a different test case for a different path constraint obtained by negating the fourth constraint. Running the program with the new input *"goo!!"*, would force a different path to be followed that is needed to execute the error. This process would be repeated making it much faster to activate the bug when compared to blackbox fuzzing.

Greybox Fuzzing

As previously stated, greybox fuzzing is a mix between both of the previously described fuzzers. We will be using the AFL fuzzer, presented in more detail in the next section, as an example. Initially, AFL applies instrumentation to be able to identify when a program executes new branches. AFL starts with an input derived from a test case given by the user and then proceeds to mutate it as it hits new branches. This is not as fast as a whitebox fuzzer, but requires less overhead as well, possibly being faster since whitebox fuzzers do not escalate well.

2.4 AFL

AFL, or *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.

AFL verifies the behaviour of the target software against incorrect data inputs. The typical bugs that can be found are:

- Faulty Memory Management
- Assertion Violations
- Incorrect Null Handling
- Bad Exception Handling
- Deadlocks
- Infinite Loops
- Undefined Behaviours

Problems like deadlocks and infinite loops can be detected by setting timeouts for the execution of the program. To detect memory issues, additional software like AddressSanitizer [29] or UndefinedBehaviorSanitizer [6] can be used to discover erroneous patterns of execution.

2.4.1 AFL description

A key innovation behind AFL is the use of coverage information obtained during the execution of the previously generated testing inputs. It is able to do this thanks to 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 instructions are added to each basic block 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 presents a simple way, with a modest performance impact, to identify new paths in the program and to have the ability to find the edges have been passed on the program.

The instrumentation is injected by a companion tool that works as a drop-in replacement for compilers, which slightly modifies the *gcc* and *clang* behaviour.

The instrumentalization code can be divided into two sections, the Trampoline code and the Main Payload code.

Trampoline: This specific assembly code is added at the beginning of each basic block in the target software, with the objective of being run when the basic block is executed. The goal is to give to every basic block a randomly generated unique 8 byte identifier. Compilers usually decompose programs into their basic blocks as a first step in the analysis process. A basic block is a straight-line code sequence with no branches in except to the entry and the exit. Algorithm 1 demonstrates how the trampoline code functions.

Algorithm 1: Trampoline

Result: AFL trampoline Instrumentation

```
1 set RANDOM_ID_OF_BLOCK
2 call MAIN_PAYLOAD;
```

The value *RANDOM_ID_OF_BLOCK* is generated randomly and is supposed to be unique in a probabilistic manner, meaning that two blocks may share an identifier.

AFL uses a shared memory *shared_mem*, with a default size of 64 kB, to track the application edge coverage. Every byte set in the output map can be thought of as a hit for a particular (branch_src, branch_dst) tuple in the instrumented code. The random identifiers of the basic blocks are employed to compute the key associated to the edge in the bitmap. Given an edge from A to B, AFL computes the key as $A \oplus (B \gg 1)$. In practise, collisions happen sporadically, with one study conducted by S. Gan et al. [14] finding that the impact of the hash collision can be significant with extremely large programs. This causes errors in the bitmap processing that leads to a loss of accuracy of edge coverage, meaning that AFL will think it has explored an edge, while in reality it has not.

Main Payload: Enables the increment of any edge hit in the *shared_mem*

bitmap. It also performs the necessary setup that is required to run the program with the instrumentalization and starts and maintains the *forkserver* process.

The first time the program under test executes, it will not have initialized the shared memory variable or the global area pointer, which is used to store the address of the shared memory region. Hence, AFL needs to initialize global_area_pointer by calling AFL_SETUP. Then, it runs AFL_SETUP_FIRST that simply attaches the shared memory segment to the variable (through *shmat*).

Algorithm 2: Main_Payload

```
   Result: AFL Main Payload
1  cur_location = RANDOM_ID_OF_BLOCK
2  if first_time_in_block then
3  |  call AFL_SETUP
4  |  if first_time_overall then
5  |  |  call AFL_SETUP_FIRST
6  |  |  call AFL_FORKSERVER
7  call AFL_STORE
8  return
```

The main function described in Algorithm 2 is used as the logic for the instrumentalization. Every time a basic block is run in the program under test, it will call the Main Payload. From there, the execution will depend if it is the first time this block is reached or if it is the first time any block is executed overall.

Algorithm 3: Main_Payload Setup and Storage

```
   Result: AFL Main Payload data initialized and Edge stored
1  Procedure AFL_SETUP
2  |  if ¬ have_global_area_pointer then
3  |  |  global_area_pointer = get_global_area_pointer()
4  Procedure AFL_SETUP_FIRST
5  |  save_everything()
6  |  shared_mem = shmat(FLAGS, global_area_pointer, AFL_SHM_ENV)
7  Procedure AFL_STORE
8  |  shared_mem[cur_loc^ prev_loc ] ++
9  |  prev_location = cur_location >> 1
```

In Algorithm 3 we simply take care of the necessary arrangements required to work with the instrumentalization and the process that is used to actually mark an

edge as hit.

Algorithm 4: AFL Main Payload Forkserver

Result: AFL Forkserver activation and block passed writing

```
1  write( OK )                                   // tell parent we are ok
2  read( GO )                    // wait for command from parent to go
3  fork()
4  if child_pid then
5  |  return
6  if parent_pid then
7  |  write(child_pid)              // write the child_pid to the parent
8  |  wait_status = waitpid()          // Wait for the child to finish
9  write(wait_status)                              // send parent result
   (finished/crashed/timed_out)
10 jmp AFL_FORKSERVER                // loop back and wait for 'go'
```

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.

That is where the forkserver comes up, as described in Algorithm 4. It 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.

The parent process then simply calls AFL_STORE that, as previously described, simply hashes the edge and adds it to the bitmap as seen.

In summary, there will at the most be three different processes at work at the same time: one process will be the fuzzer, in this case AFL; the other will be the parent process running the forkserver that waits for commands from the fuzzer and for the child pid to finish; and the last will be the process that runs the program with the new input.

The fuzzer process will simply tell the forkserver to start the fuzzing process and

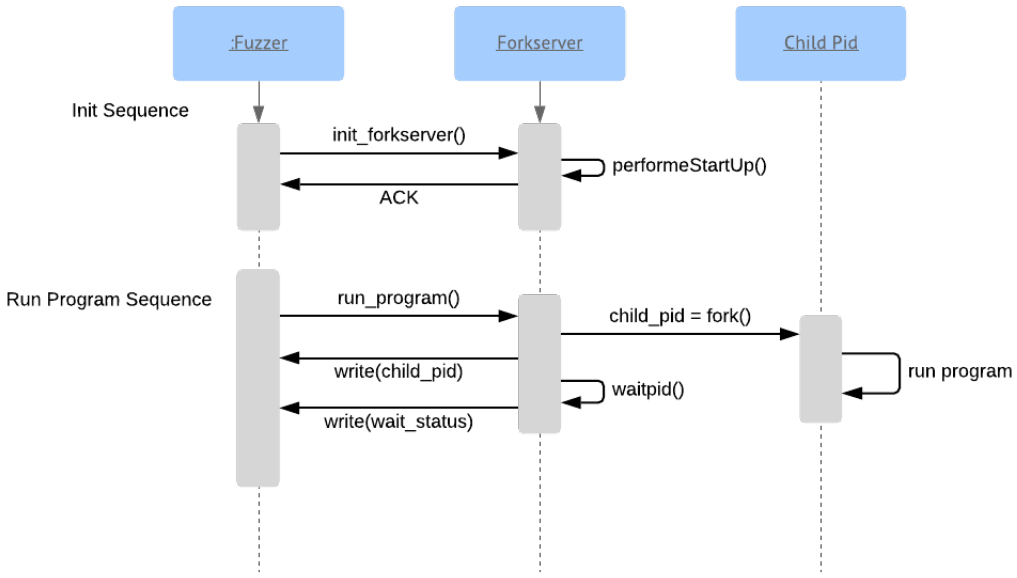

Figure 2.1: Forkserver Sequence in AFL

add the input to be fuzzed.

Algorithm 5: AFL Fuzzer process

Result: AFL Fuzzer running Forkserver returns result of run

```
1 write(input, file_to_fuzz)        // Write the input to the file to fuzz
2 write('go')                                  // Start the forkserver
3 read(child_pid)                  // Receive the pid of the child_pid
4 read(status)     // Receive child_pid status (Finished/Crashed/Timed
   Out)
5 return status
```

Then we have the process that simply waits for the go message from the fuzzer.

Then, the **child_pid** simply runs the rest of the program, marking in the *shared_mem* bitmap every edge that it passes by and then finishes, either by crashing, timing out or successfully finishing the program. More simply, Figure 2.1 demonstrates how the above works.

Fuzzing step

Now that the instrumentalization is done. We proceed to actually fuzzing the program.

The fuzzing phase works by passing through every element in the *Queue*, which is where all interesting test cases are stored. It works in a FIFO manner (First in First Out). The test cases are mutated in order to find new interesting test cases. The mutations are executed to check if they are interesting, 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*.

AFL maintains a list of the top entries in the Queue. When it finds a new path, it checks to see if the path appears to be more favourable than any of the existing ones. With the purpose of having a minimal set of paths that trigger all the bits seen in the bitmap so far. Then it prioritises fuzzing them at the expense of the rest of the entries. Hence for every byte of the bitmap AFL maintains a top-rated entry. A test case is added to the top-rated entries if the top-rated entries are not full yet, or if the contender is able to reach a certain byte faster.

Algorithm 6: AFL Main Algorithm

```
   Result: Fuzz the Target
1  Function AFL(Prog, Seeds)
2      testInitSeeds(Prog, Seeds)
3      Queue = Seeds
4      while True do
5          for testCase in Queue do
6              if ¬ isWorthFuzzing(testCase) then
7                  continue
8              score = PerformanceScore(Prog)
               // Deterministic Phase
9              mutateBitFlip(Prog, testCase)
10             mutateArith(Prog, testCase)
11             mutateInteresting(Prog, testCase)
               // Havoc Phase
12             mutateHavoc(Prog, testCase, score)
13 Function PerformanceScore(testCase)
14     score = 0
15     score += testCase.execSpeed * 0.1 > avg_exec ? 10 : 150
16     score *= testCase.bitmap_size * 0.3 > avg_bitmap_size ? 3 : 0.75
17     score *= testCase.handicap >= 4 ? 4 : 2  // how late in the fuzzing
        it was found
18     score *= testCase.depth <= 3 ? 2 : 5
19     return score
20 Function runAndMaybeSave(Prog, input)
21     runResults = runProg(Prog, input)
22     if newCoverage(runResults) then
23         addToQueue(input)
24     if isCrash(runResults) then
25         addToCrashReport(input)
26     else if isTimeout(runResults) then
27         addToHangs(input)
```

AFL logic can be described with the function help *AFL*, presented in Algorithm 6 The main functions included in the algorithm are described below:

- *testInitSeeds*: in order to test the functionalities of the SUT, and therefore test different code paths, the initial test cases (which are called seeds) must not crash or timeout. In other words, they should be valid inputs to the SUT so that further test cases can be generated from them.

- *isWorthFuzzing*: determines if the test case is worth fuzzing. A test case is worth fuzzing if it has not been fuzzed yet and it was deemed as favoured. If it has been fuzzed, it will only be fuzzed if no other favoured have been found. A test case is marked as favoured if it is among the top-rated test cases in the Queue, with the rating coming from the function *PerformanceScore*.

- *PerformanceScore*: checks the execution results and details. The number of mutation inputs to be produced is determined by the performance score given. The score returned is based on a variety of factors, such as the execution speed, bitmap size and handicap of the seed. The handicap is proportional to how late the seed appears in the fuzzing process. It defines an initial score based on the execution time, then the score increases significantly based on the size of the found bitmap, if it is bigger than the average bitmap found in the software under test it results in a bigger score. AFL gives higher importance to test cases that cause the execution of deeper paths as they have a higher chance to reveal information about the program that normally is not discovered by traditional fuzzers.

- *runAndMaybeSave*: runs the program with the test case and checks if it has new coverage. If it does, the test case is added to the Queue. After this, if the test case causes a crash to the program it adds it to the CrashQueue, and if it causes a hang or a timeout, it adds it to the HangQueue.

- *addToQueue*: This function simply appends to the queue the new test case.

- *newCoverage*: Compares the *shared_mem* bitmap that results from executing the SUT with a bitmap that aggregates information about all previous edges observed in the past, and if they differ it returns true, otherwise returns false.

- *addToCrashReport*: Adds the test case to the crashQueue and adds information about the crash. More specifically when in the mutation stage was it found.

- *runProg*: Runs the program and returns the result of the execution. If the program crashes or hangs it returns a specific error code, if it finishes normally it returns no fault.

The mutation functions present in Algorithm 6 are described in Algorithm 7.

Algorithm 7: AFL Deterministic Mutation Algorithm

```
   Result: Mutate the seeds deterministically and run the mutations
 1 Function mutateBitFlip(Prog, seed)
 2     for num_bits in [1,2,4] do
 3         for i in 0 to LENGTH(seed) do
 4             for t in num_bits do
 5                 flip_bit(seed, i + t)
 6             runAndMaybeSave(Prog, seed)
 7             for t in num_bits do
 8                 flip_bit(seed, i + t)        // set the bits back to normal
 9 Function mutateArith(Prog, seed)
10     arith_val[] = ARITH_MAX_8bits, ARITH_MAX_16bits,
        ARITH_MAX_32bits
11     for index in 0 to LENGTH(arith_val) do
12         for arith_max in arith_val[index] do
13             for i in 1 to arith_max do
14                 seed += i
15                 runAndMaybeSave(Prog, seed)
16                 seed -= i * 2
17                 runAndMaybeSave(Prog, seed)
18 Function mutateInteresting(Prog, seed)
19     interesting_val[] = interesting_8bits_vals, interesting_16bits_vals,
        interesting_32bits_vals
20     for index in 0 to LENGTH(interesting_val) do
21         for i in 0 to LENGTH(seed) do
22             for interesting in interesting_val[index] do
23                 orig = seed[i]
24                 seed[i] = interesting[i]
25                 runAndMaybeSave(Prog, seed)
26                 seed[i] = orig
```

The deterministic mutation phase consists of three different sub-phases executed one after the other at the bit level.

- *Bit flips*: The first and simplest mutation strategy is to perform sequentially ordered bit flips, switching a bit from 1 to 0 or the other way around. It does this to every bit in the test case. Then it then begins flipping bits in pairs of two adjacent bits, then it does this operation with four adjacent bits.

- *Arithmetic operations*: In order to trigger more complex conditions, AFL attempts to increment or decrement existing integer values in the test case. It starts by summing and subtracting 8-bit values. Once it has finished this part, it then proceeds to add and subtract 16-bit values, and then 32-bit values operations are performed in both endian representations.

- *interesting values*: The last deterministic stage relies on a hard-coded set of integers chosen for their elevated likelihood of triggering edge conditions in normal code (e.g., -1, 256, 1024, MAX_INT - 1, MAX_INT). The fuzzer uses a step-over of one byte to sequentially overwrite existing data in the test cases with one of the approximately two dozen ”interesting” values (the writes are 8-, 16-, and 32-bit wide in both endian representations).

The havoc mutation phase has two sub-phases:

- *Havoc*: Is a cycle with stacked random tweaks that can vary from only modifying one byte to one double word. The random mutations attempted in this stage include bit flips, overwrites with random and ”interesting” integers, block deletion, block duplication and an assortment of dictionary-related operations.

- *Splicing*: This is a last-resort strategy that only occurs if all of the previously described deterministic mutation strategies and the previous Havoc stage have not wielded any interesting, or favourable, results. It involves picking two distinct test cases from the queue that differ in at least two locations and splicing them at the random location.

2.4.2 AFL issues

Since AFL uses a simple hash of the edge as a key to the bitmap, there is a risk of collisions. This would mean that AFL would not be able to distinguish between two edges with the same hash, causing a coverage inaccuracy. S. Gan et al. [14] studies how much of a negative impact the hash collision issue has in the coverage of a program. They also propose an algorithm to resolve the hash problem and proposed new seed selection policies. It is observed that the impact of the hash collision in an application with 260K edges occurs over 75% of times. They conclude that the reason for this is the size of the bitmap and the algorithm used for hashing, since in an 64KB bitmap it is only possible to save information about a maximum of 64K edges.

Coverage inaccuracies can blur fuzzers ability to find bugs, causing certain paths to go undiscovered by the fuzzer, making the fuzzer explore other paths that might not contain vulnerabilities. There are three types of fuzzer granularity, block coverage, edge coverage and path coverage.

Chapter 3

PandoraFuzzer

This chapter describes in detail the solution developed as well as the logic behind each and every one of its components. Section 3.1 explains the problem the thesis is attempting to solve. The main emphasis of this section is to detail the reasoning behind the development of the tool and the issues found.

The next section, 3.2, goes into detail about the alterations that were made to AFL, mainly on the logical part of the architecture as well as alternatives that were considered and eventually dismissed. Section 3.3 provides the general architecture of the solution, explaining key module interactions, and presents the changes that were made to the original fuzzer, AFL.

Finally, Section 3.4, describes the modules, showing how they relate to the previously identified issues.

3.1 Tool Reasoning and Issues Found

During the creation of the tool, several problems had to be solved in order to implement the correct functionality. All the problems enumerated below occurred with the intent to build a tool that learns how to best test a given variant of a program 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.

3.2 Main AFL Differences

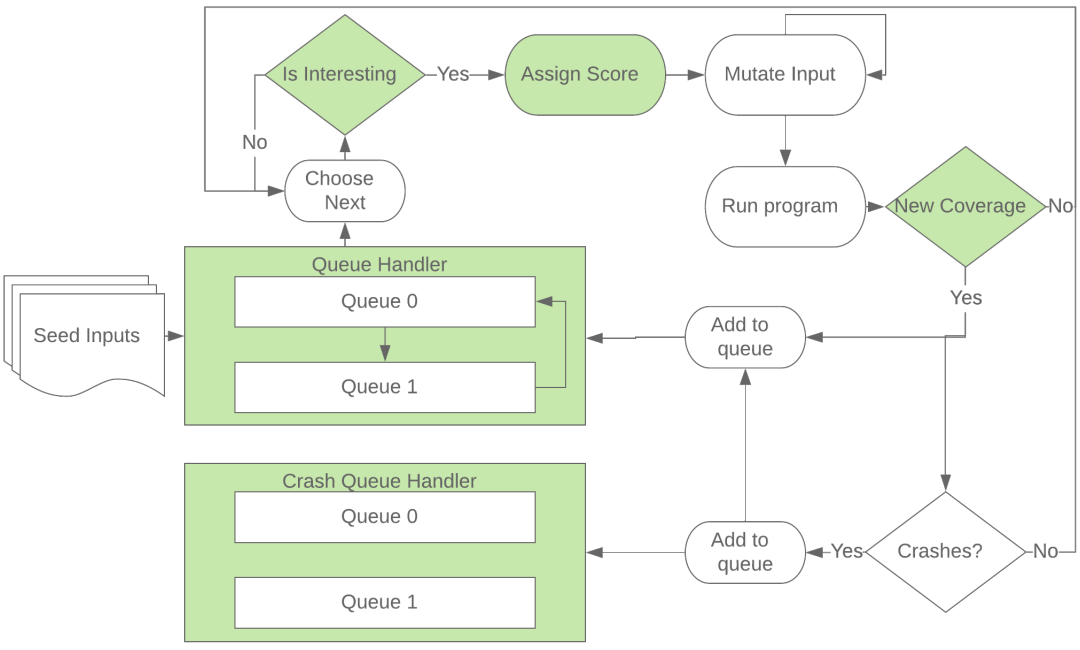

Figure 3.1: Proposed fuzzing procedure architecture of the solution

The solution presented in this chapter is based on the highly effective and popular greybox fuzzer AFL. To better describe the differences between the two fuzzers, AFL and PandoraFuzzer, this chapter describes 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.

Figure 3.1 highlights in green the modules that were modified in the original AFL architecture to create our own fuzzer solution.

3.2.1 Program Instrumentalization and Multiple Forkserver Usage

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 3.1. 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 (i,e., a hash) of the contents of each basic block.

The computation of the hash of a basic block consists of three steps. First, the fuzzer captures the basic block assembly code, obtaining the operations that influence the behaviour of the basic block and what it does. It then proceeds to create a string with every important operation concatenated. Finally, the fuzzer hashes the resulting string in a way that best distributes the resulting identifiers through a map of limited size, to minimize hash collisions when the basic blocks are fundamentally different. Otherwise, the rest of the program instrumentalization is done much like AFL. Unlike AFL where there is a single forkserver for a program, we have multiple forkservers for the various program 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.

3.2.2 Multiple Program Transition and Usage

By definition, single program vulnerability detection tends to lack the motivation to explore other programs besides the SUT. This is not a luxury that PandoraFuzzer has because its focus is on fuzzing multiple program variants. 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. This is described in more in detail in the next section.

More than one solution was considered of course. 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. This was not the case, however, when this solution was implemented. 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.

3.2.3 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, PandoraFuzzer might fuzz more than a program. Hence, there is a need for multiple queue management in the fuzzing mechanism.

Multiple queues however, is not the only solution that is available for consideration. For example, an alternative solution is to resort to a single bigger queue, which would contain more complex information to allow all inputs and crashes to reside at a single place. This could bring the benefit of simpler programming when developing the fuzzer. On the other hand, the solution was put aside on the premise that it would create a much larger queue which could cause slower queue operations. For example, information about all crashes, independently of the SUT, would go into a single queue, making it harder to identify which crashes belong to which program variants and delaying search operations.

3.2.4 Interesting Program Code Block Identifier Retrieval

In order to be able to identify which basic blocks the current input triggered, PandoraFuzzer needs to have a way to be able to track code coverage in any given program. As such, the implemented 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. This is much the same as what was implemented in our solution. 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.

One approach that was considered was 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. In theory, this would allow a better search of specific locations in the program variant. In practice, however, since we already focus our efforts on fuzzing paths that were never seen before, this would be redundant. Another issue of this solution was that in order to be able to discover the areas of the code that were deemed interesting, the tool would have to use node coverage instead of edge coverage to better identify that specific area. This would incur a significant penalty in the way we get information about processed paths during the execution of a given input.

3.2.5 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, the tool must possess a mechanism that allows it 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 that was developed is derived from a very simple concept. Whenever the tool 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 tool 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.

3.3 Architecture

This section presents the architecture that was used to develop the PandoraFuzzer, along with the key modules and key concepts necessary for proper functioning. The

architecture presented here focuses on the development of a solution that could solve all the issues described previously. As such, it allows for a multitude of programs to be fuzzed at the same time and, as previously described, 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. The architecture presented here also 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 solution also lets the tool be able to function normally even when fuzzing a single application.

We divide the architecture into two different and separate parts that when combined together solve the issue presented in this thesis. The architecture is divided into the *Instrumentalization Procedure Architecture*, which displays the logic behind the instrumentalization procedure that is done, and the second architecture is the proposed *Fuzzing Procedure Architecture*, which fuzzes the various program variants.

The main modules included in the Fuzzing Procedure Architecture are:

1. **Queue Handler** - contains, manages and interacts with the queues that represent the exploration state of the program. Each program variant will be represented by a Queue in the Queue handler. Each queue contains 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.

2. **Next Test Case Selection** - chooses the next test case to fuzz and mutate. Our solution functions like AFL when selecting the next element. To simplify, the tool evaluates and marks each test case as interesting or not interesting; when selecting a test case, it 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, the tool considers a test case interesting if it uncovers new edge information among any program variant. 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.

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. This is true for both AFL and our solution, with the score being calculated much the same way as AFL. 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.

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 tool has a crash queue for every single program 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.

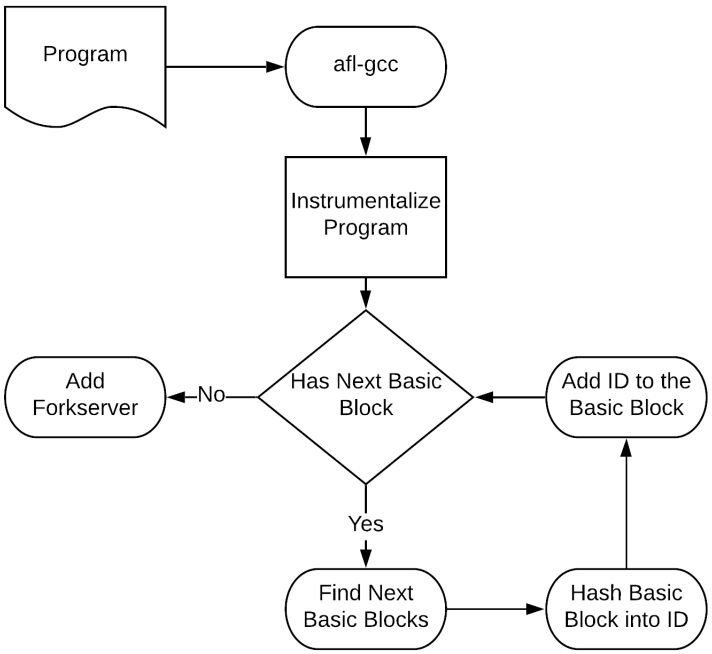

Figure 3.2: Proposed Instrumentalization Procedure Architecture of the solution.

The modules that compose the Instrumentalization Procedure Architecture are:

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 is being generated. In Figure 3.2 are represented the main steps a program goes through while being compiled by afl-gcc.

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. **Generate Identifier of Basic Block**: A major difference between AFL and our tool is the way the basic bock identifier is generated. Basic block Id generation in AFL consists of simply creating a random value, while the tool developed generates the identifier by virtue of hashing the contents of the basic block itself.

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

5. **Forkserver**: The forkserver itself, a big and important part of AFL, is left unchanged in our tool.

3.4 Main Modules

This section, goes into more detail how the main modules work and interact with each other.

3.4.1 Instrumentalization Procedure

As stated before, the solution is comprised of two phases, the instrumentalization phase and the fuzzing phase.

The goal of the instrumentalization phase is to simply facilitate the fuzzing phase by providing information about the structure of the program under test. As such, the fuzzing phase can decide which blocks to target in the future and which blocks are less interesting. The instrumentalization phase also allows for faster execution rate by means of the forkserver, explained further down.

Generation Of Basic Block Identifier

Structural information of any given program is highly important both in this solution as in AFL. The outputs of this module are used by many of the other modules from the fuzzing process.

In AFL, the generation of the basic block identifier is extremely simple, being generated randomly. This is enough in AFL case because it is not looking for shared structure information across program variants.

The tool we propose is, however, looking for the ability to differentiate between two programs functionalities. This is done at the basic block level. The generation of the identifier is based on the contents of the basic block itself. In essence, this

means that, if done correctly, 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.

To be more precise, the generation of the identifier firstly divides the assembly code of any basic block into lines. It then removes all unimportant lines from the assembly code, such as line information or labels. Afterwards, it concatenates all important info into a single line and proceeds to hash it. 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.

The implementation of the generation of the basic block identifier is described in greater detail in Chapter 4.

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.

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.

3.4.2 Fuzzing Procedure Architecture Modules

Queue Handler

It is of the upmost importance the ability to maintain information about multiple test cases in an efficient manner.

Each queue that the Queue Handler possesses is implemented using an hash map data structure that allows it 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 identifier 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. The stored value has an

8-bit capacity which is a limitation. One issue that occurs due to this limitation is that any basic block that is passed more than 255 times (the maximum value that can be stored with 8 bits) simply flips around to zero, which is a miss-representation of the value itself.

An added benefit of the implementation of multiple queues is the organisational aspect it provides to the tool. 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 mentioned in this section 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.

Score Assignment

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.

Interesting and Uninteresting Test Case Partition

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. This is done much like AFL with one small difference. Our tool deems a test case interesting not only if it triggers new coverage in the program under test, but also if it uncovers new information in the variants of the program being tested.

The selection of the next test input to fuzz is a highly important subject that can make or break a fuzzer. If the tool 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.

Input Mutation

Input mutation is the cornerstone of AFL. This mechanism explores the paths of the program under test trying to maximise speed. It consists of two phases, deterministic mutation and havoc mutation, as previously described in the related work chapter. This process remains largely unchanged in the solution presented here.

Crash Queue Handler

The Crash Queue Handler is a simple addition to manage and facilitate all crash reproduction. It allows the fuzzer (and the user) to better know what test case crashed which program. This manager is implemented in the tool as a simple queue per program, to facilitate the simple addition of test cases to the queue itself.

Chapter 4

PandoraFuzzer Implementation

This chapter presents the implementation of the PandoraFuzzer that supports the approach we described in Chapter 3, as specified in Figure 3.1 and in Figure 3.2. We start by describing the implementation of the instrumentalization phase in Section 4.1, followed by explaining the fuzzing phase in Section 4.2.

The prototype we developed is based on AFL, an open-source greybox fuzzer. Our tool allows for the detection of the same types of vulnerabilities as AFL. PandoraFuzzer was implemented mostly in C but has a single component in the instrumentalization phase that was implemented in assembly code. This is the code that is injected into the programs to be tested.

Figures 4.1 presents how the instrumentalization phase operates. The fuzzing phase is divided in two parts, the setup represented in Figure 4.6 and the main fuzzing loop, shown in Figure 4.7. The functions presented in the figures are in different colours depending on: i) if they were created from scratch, represented in green; ii) if heavily modified, shown in blue; iii) not modified or only modified slightly, if displayed in black.

4.1 Instrumentalization

This component was mostly implemented in C. The only sub-component that was implemented in assembly is the code that is injected in the binary of the program under test. This sub-component is inserted by *gcc* or *clang*, allowing the collecting of coverage information and supporting the forkserver. Once we have passed through each line of a basic block, the tool proceeds to generate the identifier of the basic block based on the contents of all the interesting lines. This phase is executed once for each program under test.

Figure 4.1 shows how the instrumentalization phase works, explaining each step. The calculation of the identifier of a basic block is described in more detail in Section 4.1.1.

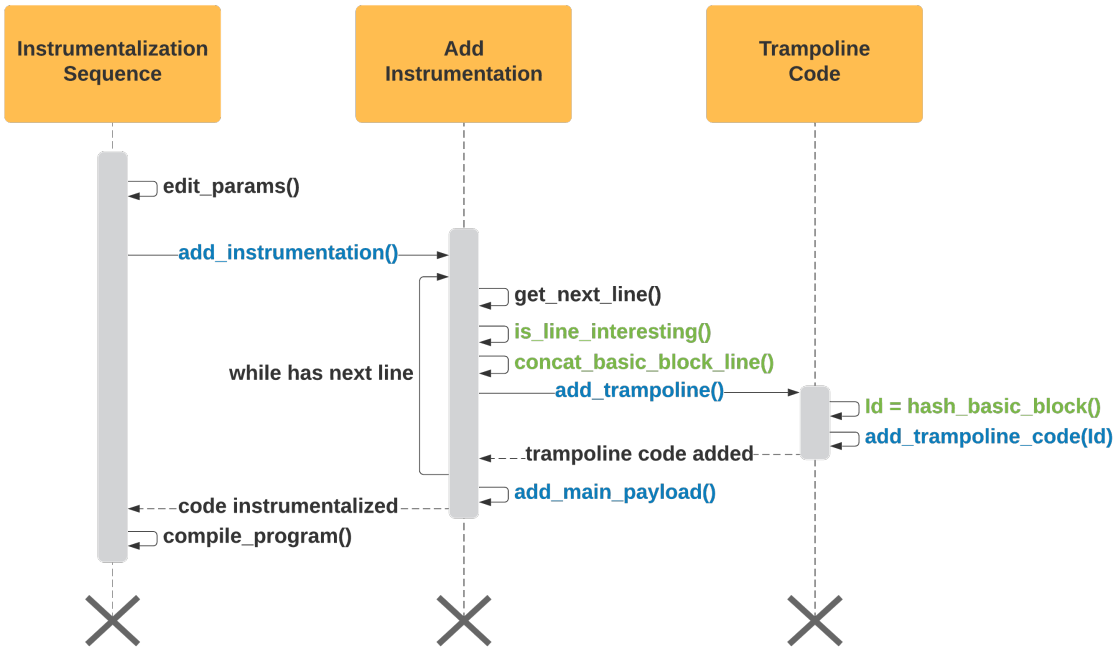

Figure 4.1: Instrumentalization Sequence in PandoraFuzzer

- *edit_params*: the goal of this function is to examine and modify the parameters passed to the program to be tested.
- *add_instrumentation*: processes the input program and generates the modified file by inserting the trampoline instrumentation in all appropriate places.
- *get_next_line*: goes through every line of the resulting assembly of the program to be instrumentalized.
- *is_line_interesting*: checks if the line is useful for the functionality of the basic block. A line is not interesting if for example it simply adds a label. Uninteresting lines are not considered further for processing.
- *concat_basic_block_line*: concatenates the important information contained in the line into a single string that will be used later to generate the identifier.
- *add_trampoline*: Starts the injection of the trampoline code into the binary of the program under test.
- *hash_basic_block*: hashes the concatenated lines to generate the identifier of the basic block. The function is Pearsons hashing with a 16-bit size, which is used to guarantee the best possible distribution of the blocks identifiers, i.e., to generate as many as possible unique identifiers.

- *add_trampoline_code*: injects the code with the previously generated basic block identifier on the current basic block line in the program.
- *add_main_payload*: when there are no more basic blocks to consider, it adds the main payload to create a shared memory with the fuzzer to add coverage information by a given input.
- *compile_program*: compiles the resulting instrumented program with either *gcc* or *clang*, depending on the configuration chosen by the user.

4.1.1 Basic Block Identifier Generation

This section goes into more detail about the generation of the identifier, along with an example. The identifier is created in three steps: (1) remove all unnecessary information; (2) remove registers; (3) concatenate all remaining information for hashing.

As was 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.

```
    .loc 1 38
    movl    %ecx, %eax
    movl    $2, %esi
    cltd
    idivl   %esi
    leal
    1(%rax),%edi
.LVL7:
.LBB40:
.LBB41:
    .loc 1 11 0
    cmpl    %edi, %ecx
    jle     .L15
```

Figure 4.2: Basic block example.

Figure 4.2 shows a basic block, which will be used as an example to show the generation of the basic block identifier.

The first 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. The changes applied to the example basic block can be seen in Figure 4.3, where the labels have been eliminated.

The second step removes the registers (e.g., *eax*) because their contents are volatile. Hence, they could change across program variants. The resulting basic block can be observed in Figure 4.4.

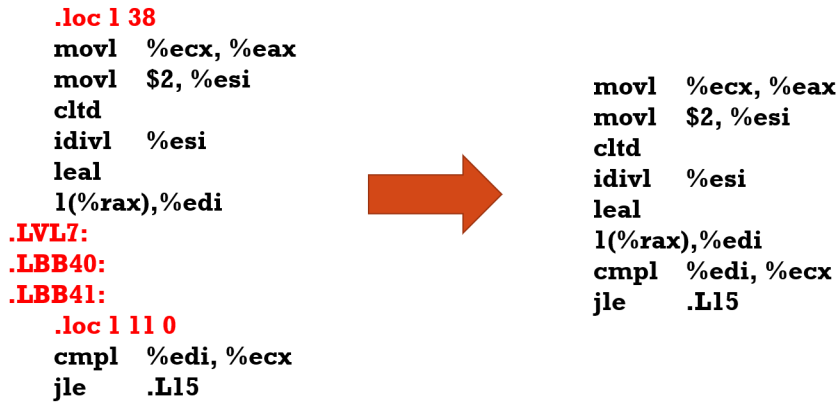

Figure 4.3: Removal of unnecessary lines of basic blocks.

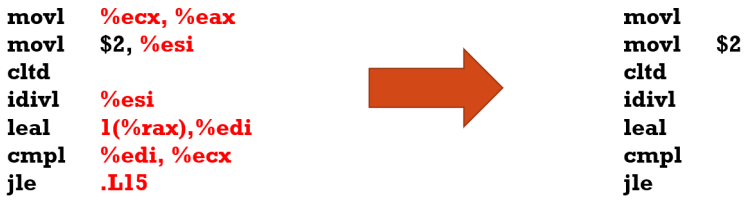

Figure 4.4: Removal of the information about registers.

The third and final step concatenates all the remaining lines and hashes them into the identifier. The final concatenated basic block to be hashed is shown in Figure 4.5.

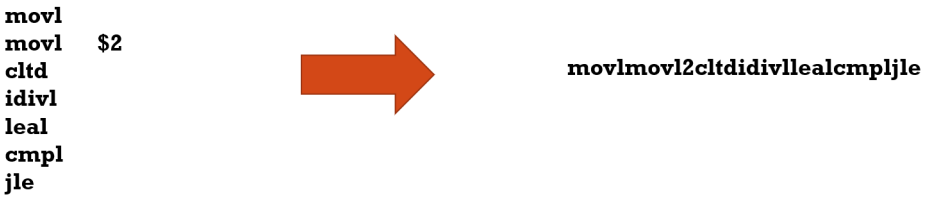

Figure 4.5: Concatenation of lines and generation of the identifier.

4.2 Fuzzing

After instrumentalizing the program under test, we start the fuzzing procedure. Figure 4.6 illustrates how the PandoraFuzzer setup is done, namely the fuzzer itself, while Figure 4.7 demonstrates the working sequence of the second half of the fuzzing procedure. In the following we present the main functions incorporated in this operation.

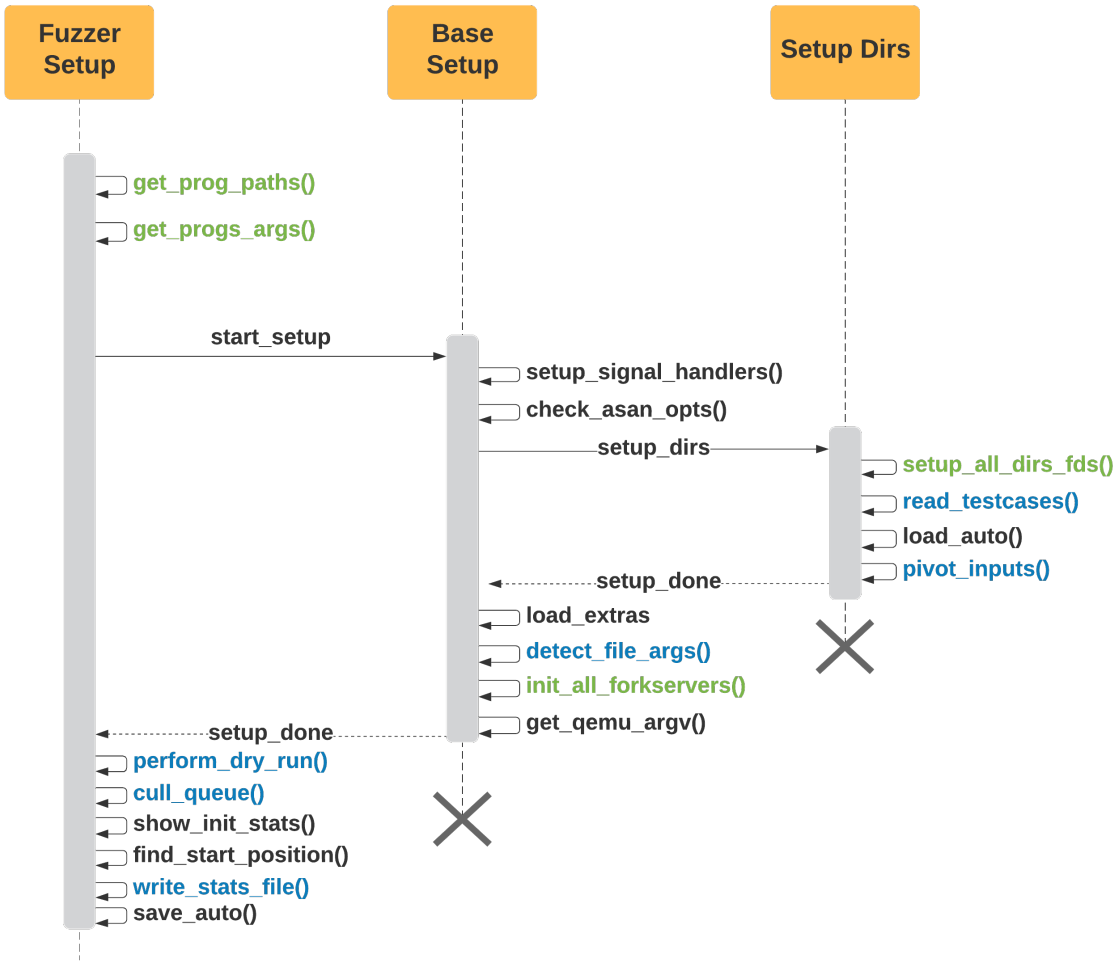

Figure 4.6: Setup Sequence in PandoraFuzzer

- *get_prog_paths*: obtains the paths given as arguments for the program variants under test. The paths are used later to execute the programs and to check if they were instrumentalized.

- *get_fuzzer_args*: obtains the fuzzer arguments, such as the input directory and the output directory. The input directory is used to store the user provided input test cases and the output directory to store all output information from the fuzzing process.

- *setup_signal_handlers*: sets up signal handlers, such as the various ways of stopping the fuzzing procedure and dealing with timeout window resizing.

- *check_asan_opts*: checks if the current fuzzing procedure is using AddressSanitezer or MemorySanitizer. These programs can be used to increase the amount of vulnerabilities found by PandoraFuzzer.

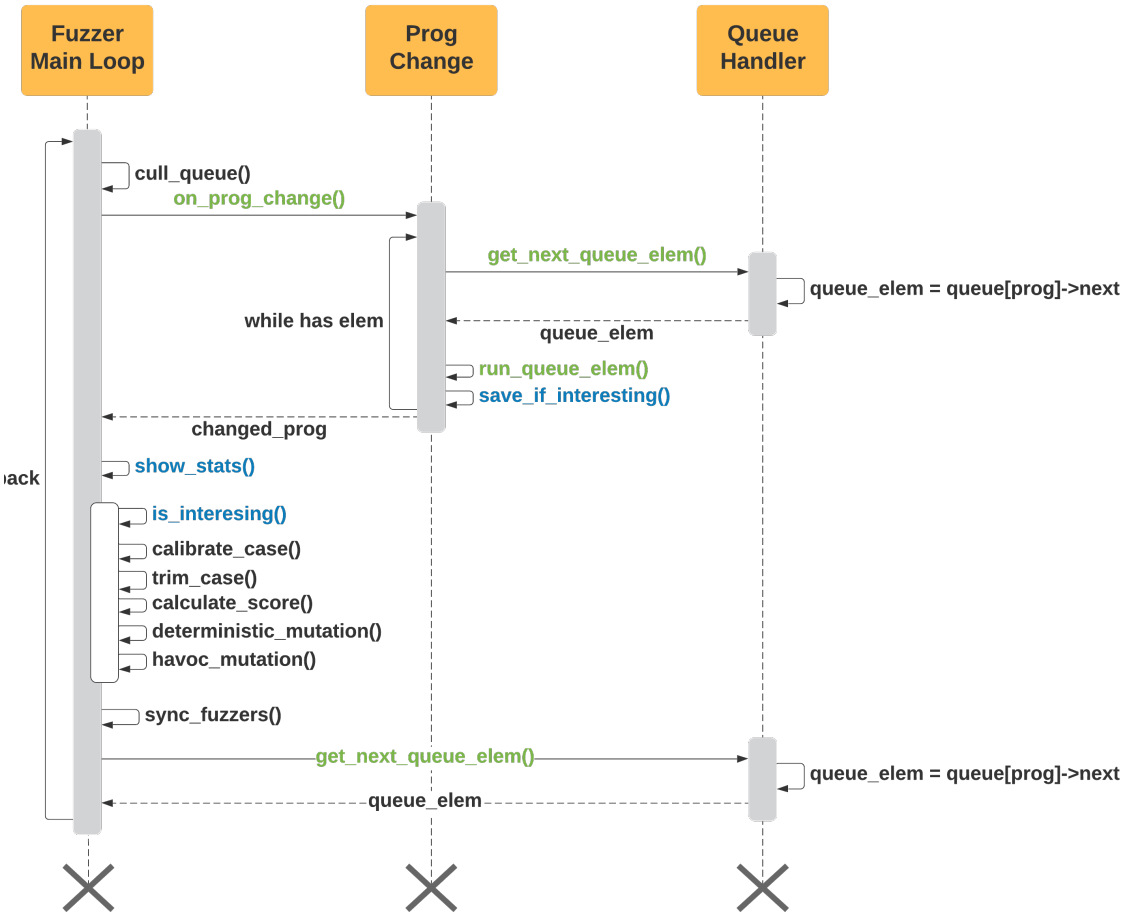

Figure 4.7: Main Fuzzing Loop in PandoraFuzzer

- *setup_all_dirs_fds*: prepares each program variant output directory where relevant fuzzing information will be saved. It checks if the given output directory already exists. If it does, it checks if it has considerable information about the previous fuzzing session and stops the setup procedure. Otherwise, it creates all directories in the output_directory and adds the plot stats information.

- *read_testcases*: reads all test cases from the given input directory, stores them in the queue for the initial program, and then queues them for testing. This function is only called at start-up. If no valid test cases are specified in the input directory, the function warns the user and stops the fuzzing procedure.

- *load_auto*: loads automatically generated extras, such as particular words that seem interesting to be included in the test cases.

- *pivot_inputs*: creates hard links for inputs test cases in the output directory by passing through the given test cases and linking them.

- *load_extras*: reads extras from the extras directory and sorts them by size. Such as a user provided dictionary.
- *detect_file_args*: checks to see if the programs receive files as command line arguments. If they do, marks where the files are supposed to go.
- *init_all_forkservers*: starts each program forkserver as well as checks each program binary to verify if it was instrumentalized. It validates the program binary by executing the program once and waits for an ACK. If the confirmation is not received in a given amount of time, it considers that the program does not have instrumentalization.
- *get_qemu_argv*: rewrites the arguments for the programs under testing for QEMU.
- *perform_dry_run*: performs a dry run on all test cases confirming that the fuzzer is running as expected. Only done with the initial input test cases and only once.
- *cull_queue*: goes over the top-rated entries from the queue and then sequentially grabs winners from previously-unseen bytes and marks them as favoured. The favoured entries are given more time during all fuzzing steps.
- *show_init_stats*: displays quick statistics at the end of processing the input directory, such as warnings if the program is slow or the test cases are too big. It displays the average execution time of the testing inputs.
- *find_start_position*: when resuming a fuzzing session, it finds the testing input to start from. This test case is the next input that was to be used in the fuzzing process.
- *write_stats_file*: updates a file with statistics about the current program being tested. The stats file contains information about the start time of the fuzzing process, the timestamp when it was last updated, the fuzzer PID, the number of cycles done, and the number of executions done along with the average execution speed. It also contains information about the total number of paths discovered, the total number of favoured paths and the max depth found in any of those paths. The stats file also contains information about the crashes that were discovered.
- *save_auto*: saves automatically generated extras.
- *on_prog_change*: called when a given time as passed. It passes through each queue element, checks if it has been fuzzed in this program, and if it has not, checks if it generates new program coverage.

- *get_next_queue_elem*: returns the next queue element to test in a given program.
- *run_queue_elem*: runs the specified queue element in the current program under test. This function is used to run any given queue element in any given program so it can look for new code coverage.
- *save_if_interesting*: checks to see if there is new code coverage was attained by checking if a new edge was executed. If so, it adds the given input to the program queue.
- *show_stats*: called every given program execution. Shows the fuzzing information to the user, such as for how long the program has been running and which program is currently under test. Also gives information about the total number of unique crashes, unique hangs and unique timeouts as well as the time passed since they last occurred.
- *is_interesting*: checks if the test case is deemed interesting to fuzz, such as if it allowed the discovery of new code coverage or if there are no other interesting test cases.
- *calibrate_case*: this is done when processing the input directory to warn about problematic test cases early on and when new paths are discovered to detect variable behaviour.
- *trim_case*: reduces the test case to the shortest possible size without influencing the execution trace of the said test case.
- *calculate_score*: calculates the score of a given test case. This score will influence the amount of time the test case is fuzzed.
- *deterministic_mutation*: deterministically mutates the test case. Only done once per test case.
- *havoc_mutation*: randomly mutates the test case according to its score.
- *sync_fuzzer*: when multiple fuzzing processes are interacting with each other, this function grabs interesting test cases and shares them between the fuzzers.

Chapter 5

Evaluation of the Tool

The main measure of how good a fuzzer is when compared with another fuzzer is the number of vulnerabilities it can detect in a given amount of time. As an approximation of this value, we can count the number of unique crashes that are found. We may also take into consideration the code coverage each fuzzer can achieve as a complementary metric of the effectiveness of the generated tests.

This chapter presents the results for the evaluation process, starting with the presentation of the test environment in Section 5.1, followed by a description of the applications under test and the experimental results in Section 5.2 .

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.

5.1 Testing Setup

The main objective of evaluating the tool is to test and verify the correctness of the solution as well as the efficiency when detecting vulnerabilities in program variants.

To do this, the following questions must be answered:

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?

The PandoraFuzzer was tested and validated in a single computer provided by the faculty. The machine had 48 CPU cores and had Ubuntu 18.04 as the operating system.

The testing was performed on four different applications from 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. [21]. 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 4.1; (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.

5.2 Applications under test

In this section we present the applications that were tested to assess both the correctness and the efficiency of PandoraFuzzer.

5.2.1 Binutils Applications

Table 5.1 provides relevant important information about the binutil 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 version 2.25 and 2.26. These four applications were chosen because they have already been employed in other fuzzing studies and because they are mainstream applications used every day by a great number of people.

5.2.2 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

Binutils				
Version	Programs	LoC	Total Files	Total paths
Version 2.25	cxxfilt	5994	17	2739
	readelf	13275	3	644
	strings	5899	16	81
	size	5807	14	758
Version 2.26	cxxfilt	5816	17	2703
	readelf	14315	3	281
	strings	5895	16	80
	size	5799	14	694

Table 5.1: Information about selected binutils applications (LoC - Lines of Code)

		Testing 1 hour		Testing 8 hours	
Version	Programs	AFL	Pandora Fuzzer	AFL	Pandora Fuzzer
Version 2.25	cxxfilt	100	123	406	505
	readelf	0	0	0	0
	strings	0	0	0	0
	size	12	8	18	12
Version 2.26	cxxfilt	95	140	497	561
	readelf	0	0	0	0
	strings	0	0	0	0
	size	0	0	0	0

Table 5.2: Unique crashes observed in Binutils applications.

provide an indication of potential bugs that are triggered with particular test cases. Table 5.2 shows the results for both tools for a testing period of 1 hour and 8 hours.

Figure 5.1 and Figure 5.2 depicts the average number of crashing test cases detected by PandoraFuzzer and AFL in Binutils 2.25 for one-hour tests and for eight hours tests, respectively. As the figures indicate, PandoraFuzzer was able to identify more crashing inputs than AFL.

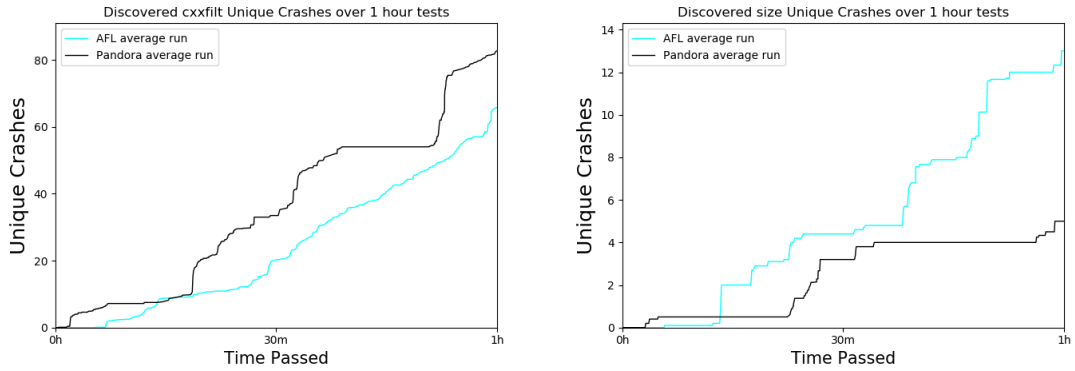

Figure 5.1: Detections over a period of 1 hour for fuzzing tests

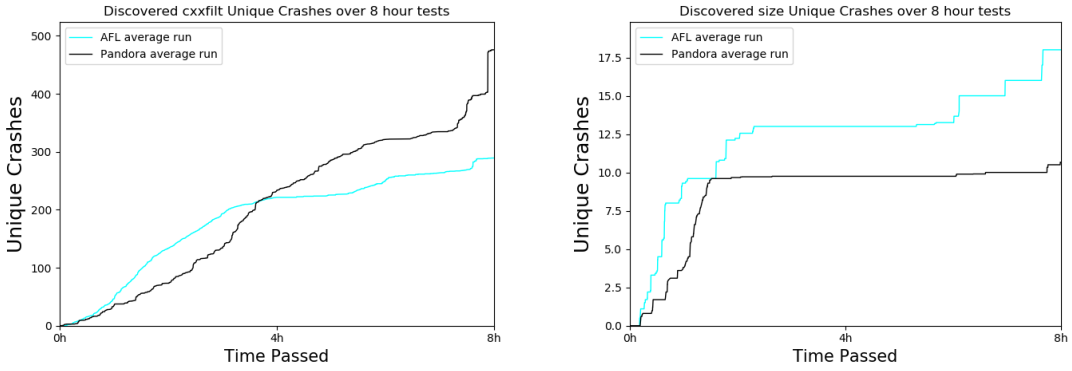

Figure 5.2: Detection over a period of 8 hours for fuzzing tests

The results presented in this section answer both the first and the last question presented in Section 5.1. The first question is answered since all types of vulnerabilities AFL found here were also found by PandoraFuzzer. The last question is answered since PandoraFuzzer managed to, on average, detect more unique crashes than AFL.

5.2.3 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 5.3 shows the total number of paths discovered by both tools for the four Binutils programs. While we can see an increase in the number of paths found by PandoraFuzzer in Binutils 2.25 that does not occur in Binutils 2.26. 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.

Version	Programs	AFL	PandoraFuzzer
Version 2.25	cxxfilt	5811	5962
	readelf	1090	1415
	strings	72	74
	size	1599	1143
Version 2.26	cxxfilt	6597	5820
	readelf	1090	1028
	strings	75	71
	size	1443	1057

Table 5.3: Maximum number of paths found in Binutils

		Testing 1 hour		Testing 8 hours	
Version	Programs	AFL	Pandora Fuzzer	AFL	Pandora Fuzzer
Version 2.25	cxxfilt	2174	1746	6157	5086
	readelf	280	412	1062	1369
	strings	66	47	74	68
	size	694	384	1555	787
Version 2.26	cxxfilt	2503	2482	6073	5122
	readelf	279	374	1068	1022
	strings	71	60	76	69
	size	667	397	1589	688

On Figure 5.3 we can see how PandoraFuzzer compares against AFL when finding new coverage information on binutils 2.25 for one-hour tests, while Figure 5.5 demonstrates the same for binutils 2.26. The results present in Figure 5.4 and Figure 5.6 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.

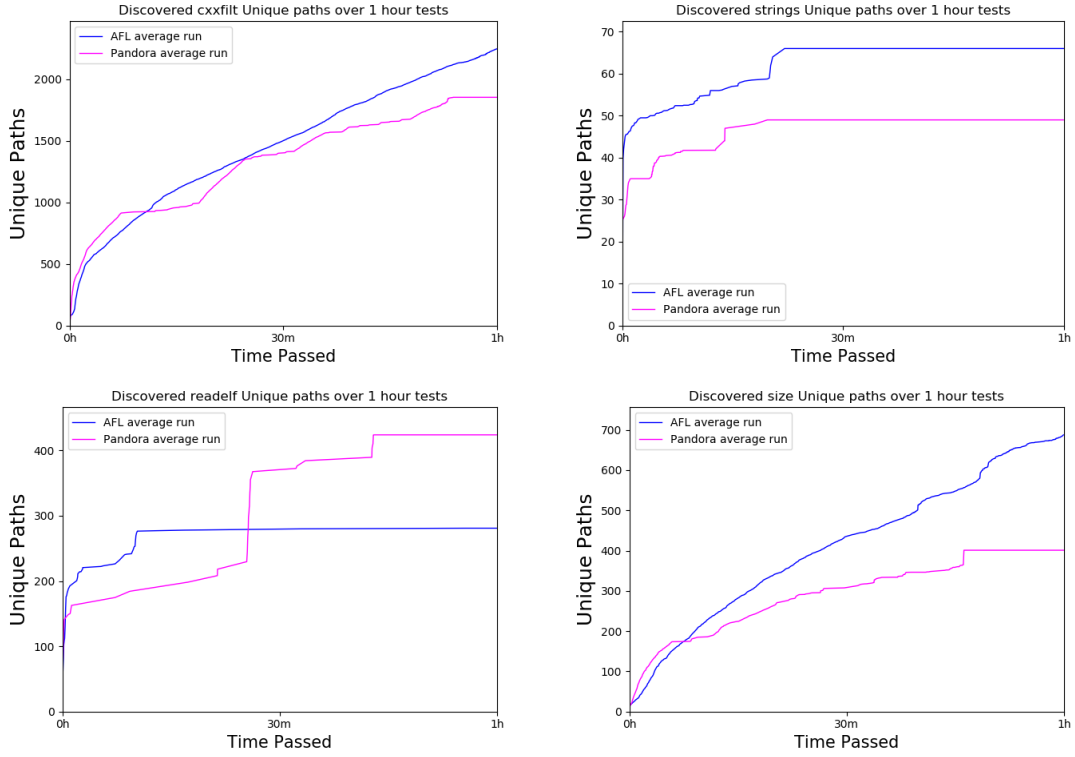

Figure 5.3: Coverage 1 hour testing for Binutils 2.25

To collect coverage information, we resorted to the afl-cov [5]. 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. The results

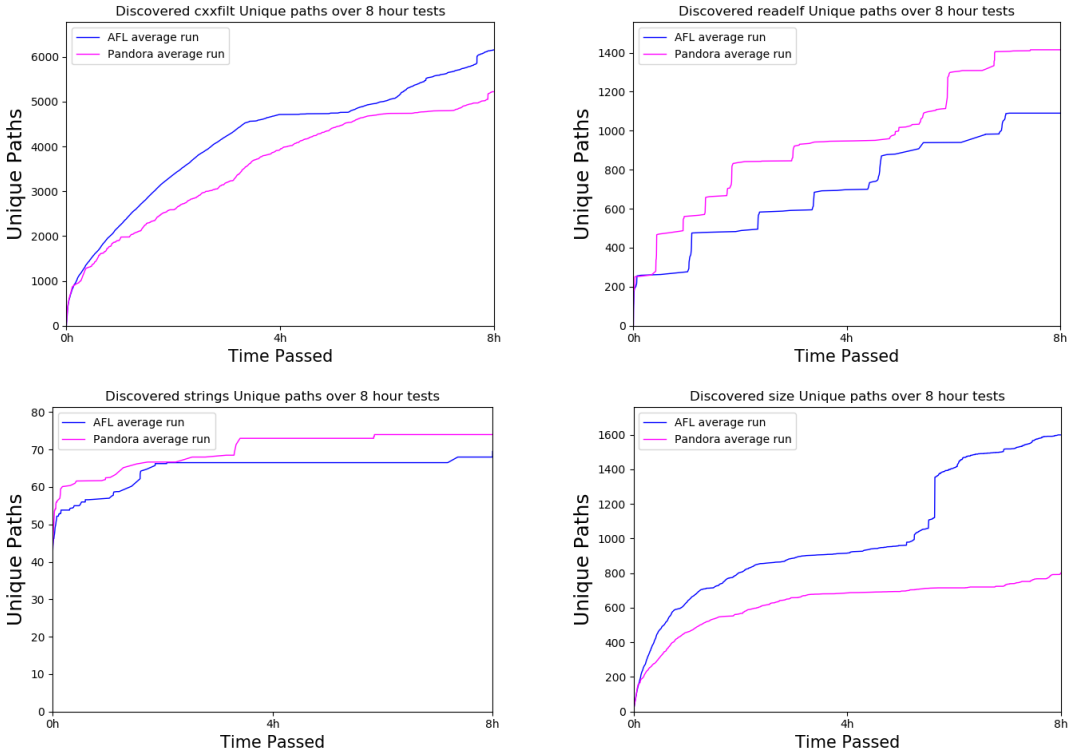

Figure 5.4: Coverage 8 hour testing for Binutils 2.25

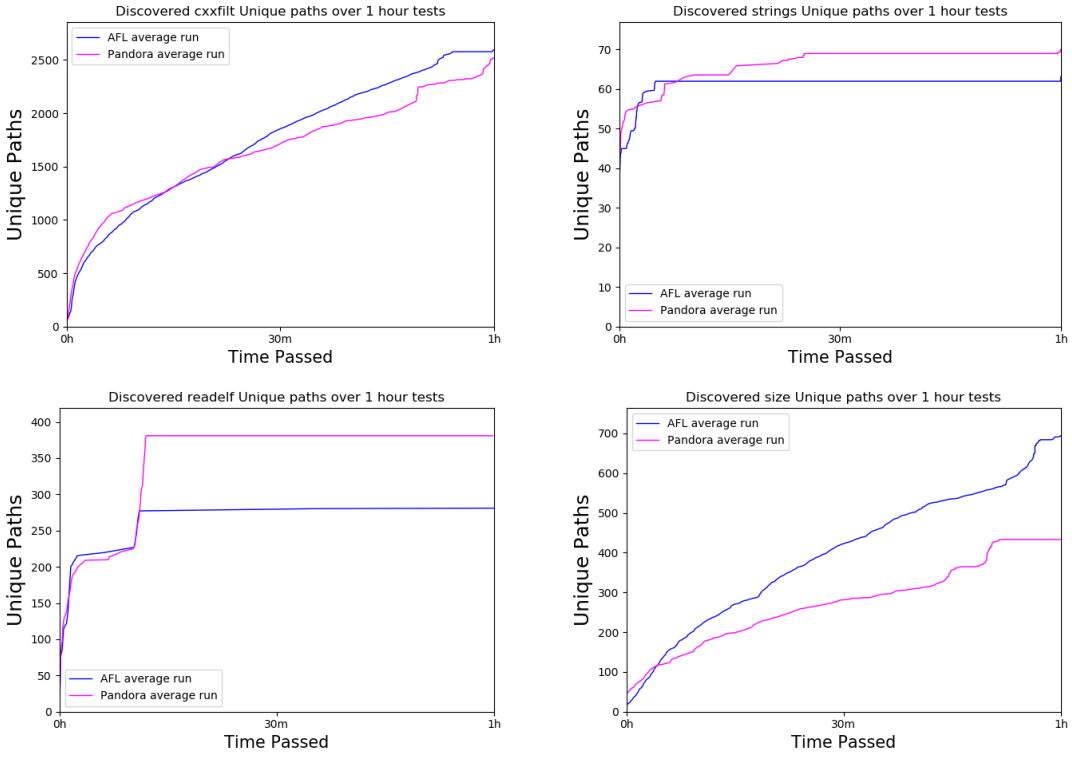

Figure 5.5: Coverage 1 hour testing for Binutils 2.26

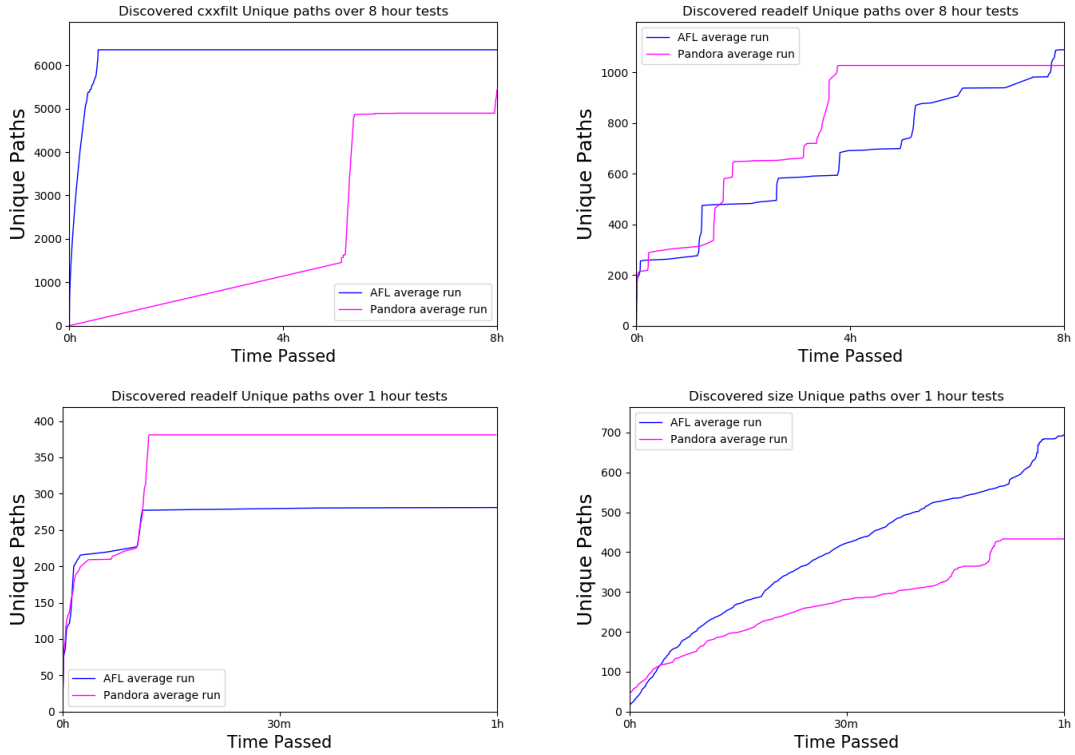

Figure 5.6: Coverage 8 hour testing for Binutils 2.26

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 presented in Section 5.1, since the focus of the fuzzing efforts changed when subjected to previously learned information.

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.

Chapter 6

Conclusion

This thesis presents an architecture and a tool for automatic vulnerability detection, utilizing the results of previous testing sessions in program variants to further boost code coverage and the number of vulnerabilities detected. The tool, called PandoraFuzzer, only works for programs that are written in the C or C++ languages.

The solution developed was built upon AFL. So, the designed architecture works as a mechanism able to identify shared functionalities among program 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 current implementation of the solution utilizes an instrumentalization mechanism that differs from AFL to be able to identify shared functionalities between program variants. Using this we can better avoid repeating tests. 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 show that PandoraFuzzer is able to detect more unique crashes and a higher percentage of total paths in a given amount of time when compared with AFL.

6.1 Future Work

Further experimentation and research is required to guarantee a certain level of performance in day to day application with a wider variety of functionalities between program variants. The instrumentalization proposed in this solution should also be researched and tested further to ensure a higher confidence on the correct functionality of this mechanism. Also, different ways to instrumentalize the programs under test should be considered, such as to take advantage of fuzzing sessions that have already been done.

Bibliography

[1] Common Weakness Enumeration. https://cwe.mitre.org/data/index.htm/. [Accessed in 30/05/19].

[2] honggfuzz. https://github.com/google/honggfuzz/. 2018, [Accessed in 21/02/19].

[3] honggfuzz Trophies. http://honggfuzz.com/. 2018, [Accessed in 22/02/19].

[4] libfuzzer. https://llvm.org/docs/LibFuzzer.html/. 2018, [Accessed in 20/04/19].

[5] afl-cov, 2018. https://github.com/mrash/afl-cov.[Accessed in 01/02/19].

[6] Undefined Behavior Sanitizer, 2018. https://developer.apple.com/ documentation/code_diagnostics/undefined_behavior_sanitizer.[Accessed in 01/02/19].

[7] M. Böhme, V. Pham, M. Nguyen, and A. Roychoudhury. Directed Greybox Fuzzing. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, CCS '17, pages 2329–2344, New York, NY, USA, 2017. ACM.

[8] 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)*, pages 122–131, May 2013.

[9] M. Böhme, V. Pham, and A. Roychoudhury. Coverage-based Greybox Fuzzing As Markov Chain. In *Proceedings of the ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1032–1043, New York, NY, USA, 2016. ACM.

[10] 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.

[11] 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*, CCS '18, pages 2095–2108, New York, NY, USA, 2018. ACM.

[12] P. Chen and H. Chen. Angora: Efficient Fuzzing by Principled Search. volume abs/1803.01307, pages 711–725, 2018.

[13] 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)*, pages 110–121, May 2016.

[14] S. Gan, C. Zhang, X. Qin, X. Tu, K. Li, Z. Pei, and Z. Chen. CollAfl: Path Sensitive Fuzzing. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, volume 00, pages 660–677, 2018.

[15] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based Whitebox Fuzzing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 206–215, New York, NY, USA, 2008. ACM.

[16] P. Godefroid, M. Y. Levin, and D. Molnar. SAGE: Whitebox Fuzzing for Security Testing. *Queue*, 10(1):20:20–20:27, January 2012.

[17] 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*, CODASPY '16, pages 85–96, New York, NY, USA, 2016. ACM.

[18] 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)*, pages 49–64, Washington, D.C., 2013. USENIX.

[19] 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)*, pages 989–1006, Vancouver, BC, 2017. USENIX Association.

[20] 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*, AISec '18, pages 37–47, New York, NY, USA, 2018. ACM.

[21] 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*, pages 2123–2138, October 2018.

[22] C. Lemieux and K. Sen. FairFuzz: Targeting Rare Branches to Rapidly Increase Greybox Fuzz testing coverage. *CoRR*, abs/1709.07101, 2017.

[23] 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, pages 18-21 February 2018*.

[24] 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*, pages 89–100, June 2007.

[25] P. Oehlert. Violating assumptions with fuzzing. In *IEEE Security and Privacy*, volume 3, Number 2, pages 58–62, March 2005.

[26] H. Peng, Y. Shoshitaishvili, and M. Payer. T-Fuzz: Fuzzing by Program Transformation. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 697–710, May 2018.

[27] 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*, pages 861–875, August 2014.

[28] SANS, CWE/SANS TOP 25 Most Dangerous Software Errors, June 2011. http://www.sans.org top25-software-errors/. [Accessed in 30/05/19].

[29] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *Proceedings of the USENIX Annual Technical Conference*, pages 309–318, June 2012.

[30] 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*, pages 138–157, May 2016.

[31] S. Sparks, S. Embleton, R. Cunningham, and C. Zou. Automated Vulnerability Analysis: Leveraging Control Flow for Evolutionary Input Crafting. In *Proceedings of the Annual Computer Security Applications Conference*, pages 477–486, Dec 2007.

[32] A. Takanen. Fuzzing for Software Security Testing and Quality Assurance. Technical report, June 2008.

[33] P. Thompson. Aflpin. https://github.com/mothran/aflpin. 2015 [Accessed in 20/04/19].

[34] V. van der Veen, N. dutt Sharma, L. Cavallaro, and H. Bos. Memory Errors: The Past, the Present, and the Future. In D. Balzarotti, S. J. Stolfo, and M. Cova, editors, *Proceedings of the Research in Attacks, Intrusions, and Defenses*, pages 86–106, September 2012.

[35] 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*, pages 511–522, November 2013.

[36] F. Yamaguchi, N. Golde, D. Arp, and K. Rieck. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 590–604, May 2014.

[37] M. Zawlewski. AFL Technical Details. http://lcamtuf.coredump.cx/afl/technical_details.txt/. 2018 [Accessed in 20/04/19].

[38] M. Zawlewski. American Fuzzy Lop (AFL) Fuzzer. http://lcamtuf.coredump.cx/afl/. 2017, [Accessed in 01/03/19].