Automatic Repair of Java Code with Timing Side-Channel Vulnerabilities

Rui Diogo Tomás Lima

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. João Fernando Peixoto Ferreira

Examination Committee

Chairperson: Prof. António Manuel Ferreira Rito da Silva
Supervisor: Prof. João Fernando Peixoto Ferreira
Member of the Committee: Prof. Pedro Miguel dos Santos Alves Madeira Adão

January 2021

Acknowledgments

I would like to thank everyone that made this project possible. I would like to thank my parents for their support and for allowing me to focus on this project and my studies.

I would like to thank professor João Ferreira for the idea behind the project and for his insights and advice to help make this project a reality. I would also like to thank professor Pedro Adão, for his critique of the first report of this project and advice to improve the project.

I would also like to thank the authors of DifFuzz for the creation of the tool that inspires this project and for their help in understanding DifFuzz to make this project a reality.

To those, and anyone not mentioned, my sincerest thanks.

Abstract

Vulnerability detection and repair is a demanding and expensive part of the software development process. As such, there has been an effort by researchers to develop new and better ways to automatically detect and repair vulnerabilities. DifFuzz is a state-of-the-art tool for automatic detection of timing side-channel vulnerabilities, a type of vulnerability that is particularly difficult to detect and correct. Despite recent progress made with tools such as DifFuzz, work on tools capable of automatically repairing timing side-channel vulnerabilities is scarce.

We propose a new tool for automatic repair of timing side-channel vulnerabilities in Java code. The tool works in conjunction with DifFuzz and it was able to repair 56% of the vulnerabilities identified in DifFuzz's dataset. The results show that the tool can indeed automatically correct timing side-channel vulnerabilities, being more effective with control-flow based timing side-channel vulnerabilities.

Keywords

Java, Timing Side-Channel Vulnerabilities, Automatic Detection of Vulnerabilities, Automatic Repair of Vulnerabilities, Security, Code Modification

Resumo

A detecção e reparação de vulnerabilidades é uma parte exigente e desafiante do processo de desenvolvimento de software. Por isso, tem havido um esforço por parte dos investigadores para desenvolver novas e melhoradas formas de detectar e corrigir de forma automática vulnerabilidades. DifFuzz é uma ferramenta de ponta para a detecção automática de vulnerabilidades ‘timing side-channel’, um tipo de vulnerabilidade que é particularmente difícil de detectar e corrigir. Apesar do progresso recente alcançado com ferramentas como o DifFuzz, há pouco trabalho em ferramentas capazes de corrigir de forma automática vulnerabilidades ‘timing side-channel’.

Propomos uma nova ferramenta para a correcção automática de vulnerabilidades ‘timing side-channel’ em código Java. A ferramenta funciona em conjunto com o DifFuzz e foi capaz de reparar 56% das vulnerabilidades identificadas no dataset do DifFuzz. Os resultados mostram que a ferramenta pode corrigir de forma automática vulnerabilidades ‘timing side-channel’, sendo mais eficaz em vulnerabilidades ‘control-flow based timing side-channel’.

Palavras Chave

Java, Vulnerabilidades Timing Side-Channel, Detecção automática de vulnerabilidades, Reparação automática de vulnerabilidades, Segurança, Modificação de Código

Contents

1 Introduction 1

1.1 Work Objectives . . . 4

1.2 Contributions . . . 4

1.3 Thesis Outline . . . 5

2 Background and Related Work 7

2.1 Timing Side-Channel Vulnerabilities . . . 9

2.1.1 Early Exit Vulnerabilities . . . 10

2.1.2 Control-Flow Based Vulnerabilities . . . 13

2.1.3 Mixed Vulnerabilities . . . 14

2.2 Automated Detection of Timing Side-Channel Vulnerabilities . . . 15

2.3 Automated Repair of Software Bugs . . . 18

2.3.1 Automated Repair Tools . . . 20

2.4 Automated Repair of Timing Side-Channel Vulnerabilities . . . 23

2.5 Automated Testing of Java Software . . . 24

3 System Architecture and Methodology 25

3.1 System Overview . . . 27

3.2 Identification of Vulnerable Methods . . . 28

3.3 Correction of Vulnerabilities . . . 29

3.3.1 Correction of Early-Exit Timing Side-Channel Vulnerabilities . . . 30

3.3.2 Correct Control-Flow Timing Side-Channel Vulnerabilities . . . 32

4 Implementation 37

4.1 Software Tools Used . . . 39

4.1.1 Refactoring Tools . . . 39

4.1.2 Automated Test Generation . . . 40

4.2 Driver Processor . . . 40

4.3 Correction of Early-Exit Timing Side-Channel Vulnerabilities . . . 53

4.4 Correction of Control-Flow Based Timing Side-Channel Vulnerabilities . . . 56

4.5 Correction of Mixed Timing Side-Channel Vulnerabilities 60

5 Evaluation **63**

5.1 Dataset Used 65

5.2 Semantic Correction 65

5.3 Vulnerability Correction 66

5.4 Dataset Examples 69

5.4.1 Invalid Examples 69

5.4.2 Unsuccessful Examples 76

5.4.2.A Faulty Examples 76

5.4.2.B Unsuccessful Correction 79

6 Conclusion **89**

6.1 Conclusions 91

6.2 System Limitations and Future Work 91

List of Figures

2.1 Overview of DifFuzz 18

3.1 Overview of DifFuzzAR 28

List of Tables

5.1 Results of tool 67
5.2 Early-Exit results 67
5.3 Control-Flow results 68
5.4 Mixed results 68
5.5 Invalid examples 69

List of Algorithms

3.1 Tool Code Overview 28
3.2 Identification of vulnerable method using *DifFuzz* driver 29
3.3 Overview of the repair process 30
3.4 Correction of Early-Exit Timing Side-Channel Vulnerabilities 31
3.5 Correction of Control-Flow Based Timing Side-Channel Vulnerabilities - PART 1 34
3.6 Correction of Control-Flow Based Timing Side-Channel Vulnerabilities - PART 2 35

Listings

2.1 Early Exit Vulnerable Code Snippet [1] 10
2.2 Unsafe Login [2] 10
2.3 Safe example of Listing 2.1 [1] 11
2.4 Safe Login [2] 12
2.5 Example of branch dependent execution time [3] 13
2.6 Safe example of Listing 2.5 [2] 13
2.7 Example of a Mixed timing side-channel vulnerability in a method 14
2.8 Safe example of Listing 2.7 15
3.1 Vulnerable method invocation example 29
3.2 Original Vulnerable Method 31
3.3 Vulnerable Method after early-exit correction 32
4.1 Example of simple Driver 40
4.2 Example of Driver with constructor after Mem.clear() 43
4.3 Example of Driver with try block after Mem.clear() 46
4.4 Example of Driver with if block after Mem.clear() 48
4.5 Vulnerable Method 54
4.6 Vulnerable Method after early-exit correction 55
4.7 Example of method with control-flow based timing side-channel vulnerability 57
4.8 Corrected version of method with control-flow based timing side-channel vulnerability . . 58
4.9 Example of method with invocations in condition to modify 59
4.10 Corrected version of method with invocations in condition 59
5.1 Vulnerable method encrypt 69
5.2 Vulnerable method loopAndBranch 70
5.3 Vulnerable method modPow2 71
5.4 Vulnerable method compress 72
5.5 Vulnerable method getRows 74
5.6 Vulnerable method hashMatches 75

5.7 Vulnerable method doGet 75
5.8 Vulnerable method isEqual 76
5.9 Vulnerable method regionMatches 78
5.10 Vulnerable method getPeople 80
5.11 Vulnerable method equals_inline 81
5.12 Vulnerable method equals 82
5.13 Vulnerable method authenticate 83
5.14 Vulnerable method of example gpt14 84
5.15 Vulnerable method of example themis_boot-tateless-auth 85
5.16 Vulnerable method of example stac_ibasys 87

1 Introduction

Contents

1.1 Work Objectives 4
1.2 Contributions 4
1.3 Thesis Outline 5

Software is increasingly more present in the world and our lives. Virtually all industries today use software in one way or another. These industries range from retail, healthcare, and finance to government and defense. The software these industries use must be as secure as possible since software defects can lead to the loss of billions of euros in revenue or lawsuits, or it can even harm the health and well-being of millions of people around the world. So, companies must regularly check their software for bugs and vulnerabilities, either running unit tests or analyzing the code.

There are many kinds of bugs and vulnerabilities. A bug is an issue that leads to an expected scenario not running, for instance trying to login with correct credentials and failing. On the other hand, a vulnerability is an issue that leads to an unexpected scenario running, for instance trying to login with fake credentials and succeeding [4]. To put it in other terms, a bug is a failure in an application that can lead to an unsuccessful use of the application by the user, while a vulnerability is a failure in an application that can lead to wrongful use of the application, causing security issues.

Bugs can be detected using software tests and the quality of the tests impacts the odds of detecting a bug. As such it is important to create many and comprehensive test cases. However, the detection of vulnerabilities can be difficult since a vulnerable application can pass all tests or even completely fulfil its correctness specification. Different types of vulnerabilities have different difficulty levels of detection. Perhaps one of the hardest types of vulnerabilities to be detected are side-channel vulnerabilities.

A side-channel is any observable side effect of a computation. The side effects can manifest themselves in several ways: for example, in the difference in computation time, in power consumption, sound production, and electromagnetic radiation emitted. A more in-depth explanation of side-channel attacks can be found in [5, 6]. Some side-channel vulnerabilities are easier to exploit than others since some require specific devices. All of these side effects can be taken advantage of to attack a system. However, most of the side-effects require that the attackers have physical access to the system they are trying to attack, since they would need to gather the information directly from the system, like measuring the power consumption, the production of sound, or the emission of electromagnetic radiation. On the other hand, side-effects such as the difference in computation time and response size do not require the attackers to be in direct contact with the systems. This enables the possibility of remote attacks, thus exposing the systems to a larger number of attackers. Detecting and fixing a side-channel vulnerability based on response size is, in general, easy, since they mainly occur in the creation of the response to be sent to the client. In contrast, side-channel vulnerabilities based on measuring differences in computation time, also known as timing side-channel vulnerabilities, can occur at multiple program points: they can occur on a simple method to compare strings, or on a large and complex parallel computation. There are multiple real-world applications that were found to be vulnerable to timing side-channel attacks. For instance, Nate Lawson et al. discovered a timing side-channel vulnerability in Google's Keyczar Library [7]. Another example is the timing side-channel vulnerability discovered in Xbox 360 [8].

Moreover, the authors of DifFuzz [1], a state-of-the-art tool for automatic detection of timing side-channel vulnerabilities, found new vulnerabilities in Apache FtpServer [9] and AuthMeReloaded [10].

As timing side-channel vulnerabilities are extremely difficult to detect, there is a great effort by researchers to develop tools and methods capable of automatically detecting these vulnerabilities [1–3]. Despite this, once vulnerabilities are found, developers must correct them manually, which in some cases can be difficult, time-consuming and prone to errors. As such, the goal of this project is to ease the correction of vulnerabilities by developing a tool capable of automatically repairing timing side-channel vulnerabilities. Even though the ideas developed are general and can be used in any language, we focus on the Java programming language since according to GitHub [11], Java is the second language with more contributors in public and private repositories and is still the most used language for enterprise applications according to other sources, such as Cloud Foundry [12] and IBM [13]. The tool developed, called *DifFuzzAR*, is designed to work in conjunction with DifFuzz [1] and we evaluate it using the same dataset that was used to evaluate DifFuzz. Although *DifFuzzAR* has some limitations, it repaired 56% of the vulnerabilities identified in DifFuzz's dataset. This shows that *DifFuzzAR* has the potential to simplify substantially the debugging process.

1.1 Work Objectives

The main goal of this project is to develop a tool for automatic repair of timing side-channel vulnerabilities in Java code. More specifically, the objectives of this project are:

1. To develop the aforementioned tool so that it works in conjunction with DifFuzz [1]. For instance, it can use DifFuzz's already defined "drivers" to identify the vulnerable methods.
2. To identify patterns and any sort of computation that lead to timing side-channel vulnerabilities.
3. To propose (and implement) algorithms capable of correcting potential timing side-channel vulnerabilities.
4. To evaluate the developed tool using the same dataset that was used to evaluate DifFuzz [1], measuring how many vulnerabilities were automatically fixed. This dataset contains examples of applications with timing side-channel vulnerabilities.

1.2 Contributions

The main contribution of this project is a software tool capable of repairing timing side-channel vulnerabilities in Java code. The tool is open-source and is available at:

https://github.com/RuiDTLima/DifFuzzAR

Moreover, this document documents patterns that can result in timing side-channel vulnerabilities. It also includes a categorization of these vulnerabilities as early-exit, control-flow based, and mixed timing side-channel vulnerabilities. A discussion on how to automatically correct this type of vulnerabilities is also included. This information might be useful for researchers interested in developing automated repair tools for fixing timing side-channel vulnerabilities.

Finally, the tool was evaluated on a dataset that is available to the research community, and it was able to repair 56% of the identified timing side-channel vulnerabilities.

1.3 Thesis Outline

The rest of this document is structured as follows.

- Chapter 2 provides the background necessary to understand and to work on this project and previous work related to this project. This includes a discussion on different types of timing side-channel vulnerabilities and existing tools capable of automatically detecting these vulnerabilities.
- Chapter 3 provides a general overview of the tool developed, starting with how the tool is structured. Then it provides an overview of the modules responsible for finding the vulnerable method and for correcting the vulnerability. This last one is divided into two parts, the correction of early-exit timing side-channel vulnerabilities and the correction of control-flow based timing side-channel vulnerabilities.
- Chapter 4 describes the development process followed to implement the tool. Here it is attempted to convey the idea behind each module and the challenges that arise when developing them. It starts by describing the module responsible for processing DifFuzz's drivers, in search of the vulnerable method. Next, the development of the modules for correcting the timing side-channel vulnerabilities is described. Finally, it is explained the tool used to read and modify the Java code.
- Chapter 5 describes the process followed to evaluate the tool and presents the evaluation results.
- Chapter 6 concludes the report and lists some of the limitations of the tool. It also presents suggestions of future work to improve the tool.

2 Background and Related Work

Contents

2.1 Timing Side-Channel Vulnerabilities 9

2.2 Automated Detection of Timing Side-Channel Vulnerabilities 15

2.3 Automated Repair of Software Bugs 18

2.4 Automated Repair of Timing Side-Channel Vulnerabilities 23

2.5 Automated Testing of Java Software 24

Since software security is such an important subject, several studies have been done to help developers optimize their software's security. For example, Meng et al. [14] analyzed Stack Overflow posts and found out that programmers are especially concerned about the implementation of security features and how to make their code as secure as possible. They show that there has been an increase in the focus of security and secure coding. They identify the five most common programming challenges to be:

1. ***Authentication:*** the correct use of authentication on their applications is one of the major concerns of developers. For example, as shown by Meng et al. [14], many users were concerned about integrating Spring security with different application servers or frameworks.

2. ***Cryptography:*** there is a great concern about key generation and usage.

3. ***Java EE security:*** in this API one of the most common challenges is the usage of declarative security and programmatic security.

4. ***Access Control:*** many programmers still do not know how to restrict or relax the access permission of a software application for certain resources.

5. ***Secure Communication:*** there is still a great confusion on how to create, install, find or validate an SSL certificate, and how to establish a secure connection.

These challenges, and the fact that many security breaches keep happening shows that programmers are not prepared to produce secure programs. This added to the fact that secure coding is a complex task, and the documentation is minimal, increases the need for security tools of any kind.

2.1 Timing Side-Channel Vulnerabilities

A problem that arises in the five categories listed above, with particular importance in cryptographic code, is the presence of timing side-channel vulnerabilities. Since this work will focus on timing side-channel vulnerabilities, it is essential to understand their characteristics and some of the most common ways they appear.

A timing side-channel vulnerability happens when a secret[1] can be learned based on the time a computation takes to complete. To put it differently, an application is vulnerable to timing side-channel attacks when the time it takes to complete a computation depends on a given secret, e.g., a password. There are multiple ways that a timing side-channel vulnerability can appear. Despite that, the most common ones revolve around cycles where the termination condition depends on the secret, or with an early exit condition dependent on the secret. They are also common when there is an unbalance in the

[1]A secret is any value, not known by an attacker, be it a password, a secret code, or any value that attackers attempt to learn. In the context of this paper, a secret is a value that the attackers are trying to discover.

control flow of code dependent on a secret. If the control flow of an application depends on a secret, and the execution time of that application depends on the control flow, then the application is vulnerable to timing side-channel attacks.

As a way to illustrate ways in which timing side-channel vulnerabilities show themselves, several examples of code snippets vulnerable to timing side-channel attacks taken from research articles and existing datasets of vulnerabilities are shown below.

2.1.1 Early Exit Vulnerabilities

Listing 2.1 and Listing 2.2 show two classic examples of early exit vulnerabilities. These listings show two examples of different ways to check if the given password is correct. Although they look different, the vulnerability is the same since in both there is an exit condition dependent on the secret, in this case the parameter “sec”. In these examples, resourceful attackers can “find” the password by trying different combinations of characters until they find the correct one. With this code, they know they found a correct character simply by the time the function takes to respond. For example, if the password was “bcdef” and the attackers tried with the password “abcde”, the function returns false, and they aren’t able to get any information. If next, they tried with the password “bbcde”, they would know that the first character of the password was correct simply because the function took extra clock cycles to return. With that, the attackers know that the first character of the password is “b”. They can now repeat the process for the other characters of the password until they find the correct one.

```
boolean pwcheck_unsafe(byte[] pub, byte[] sec) {
    if(pub.length != sec.length) {
        return false;
    }
    for(int i = 0; i < pub.length; i++) {
        if(pub[i] != sec[i]) {
            return false;
        }
    }
    return true;
}
```

Listing 2.1: Early Exit Vulnerable Code Snippet [1]

```java
boolean loginBad(String username, byte[] guess) {
    byte[] user_pw = retrievePassword(username);
    if(user_pw == null)
        return false;
    for(int i = 0; i < guess.length; i++) {
        if(i < user_pw.length) {
            if(guess[i] != user_pw[i])
                return false;
        } else {
            return false;
        }
    }
    return true;
}
```

Listing 2.2: Unsafe Login [2]

Since these examples are quite simple, their solution is also simple. Changing a couple of lines of code would correct the vulnerabilities shown. For example, when the timing side-channel vulnerability is due to an early exit, the solution is to eliminate the early exit. In Listing 2.1, all that was needed was to avoid the early exit that happens when the size of the secret does not match the size of the user-submitted value. And to add a variable to hold the value true if the secret was equal to what the user-submitted and false if they were different. However, only doing this still allows an attacker to uncover secrets. As there would still be differences in the execution times of the application, some extra changes would still be needed. These changes have only one objective, to make the application execution time as independent of secrets as possible. With that, the application will be less efficient since it will execute instructions that do not add any computational value. After all changes, the method in Listing 2.1 will be the method in Listing 2.3.

```java
boolean pwcheck_safe(byte[] pub, byte[] sec) {
    boolean unused;
    boolean matches = true;
    for(int i = 0; i < pub.length; i++) {
        if(i < sec.length)  {
            if(pub[i] != sec[i]) {
                matches = false;
            } else {
                unused = true;
            }
        } else {
            unused = false;
            unused = true;
        }
    }
    return matches;
}
```

Listing 2.3: Safe example of Listing 2.1 [1]

A similar approach can be followed for Listing 2.2. Since this snippet of code has the timing side-channel vulnerability for the same reasons as Listing 2.1, the solution will be very similar to Listing 2.3. That solution can be seen in Listing 2.4.

```
boolean loginSafe(String username, byte[] guess) {
    boolean dummy, matches = true;
    byte[] user_pw = retrievePassword(username);
    if(user_pw == null)
        return false;
    for(int i = 0; i < guess.length; i++) {
        if(i < user_pw.length) {
            if(guess[i] != user_pw[i]) {
                matches = false;
            } else {
                dummy = true;
            }
        } else {
            dummy = true;
            matches = false;
        }
    }
    return matches;
}
```

Listing 2.4: Safe Login [2]

2.1.2 Control-Flow Based Vulnerabilities

A control-flow based timing side-channel attack happens when there is a significantly slow operation that happens only when a condition is met. What this means is that an attacker can take notice of the time a function takes to return and notice that that operation is executed. In Listing 2.5, if the condition is true, two slow methods will be executed, while if the condition is false, the execution time will be much faster.

```java
BigInteger modPow(BigInteger base, BigInteger exponent, BigInteger modulus) {
    BigInteger s = BigInteger.valueOf(1) ;
    int width = exponent.bitLength();
    for(int i = 0; i < width; i ++) {
        s = s.multiply(s).mod(modulus);
        if(exponent.testBit(width-i-1))
            s = s. multiply(base).mod(modulus);
    }
    return s;
}
```

Listing 2.5: Example of branch dependent execution time [3]

A common problem with the solutions proposed for timing side-channel vulnerabilities is that they typically make the code less efficient. This happens because a common cause of these vulnerabilities is the use of methods with slow execution in conditional branches. As such, a common approach is to add a similar operation to the other branches. With that, regardless of the branch executed, the execution time will be as similar as possible. Examples of such a solution can be seen in Listing 2.6, which is a corrected form of the method in Listing 2.5. In this examples, the parameter “exponent” is a secret.

```java
BigInteger modPow1_safe(BigInteger base, BigInteger exponent, BigInteger modulus) {
    BigInteger s = BigInteger.valueOf(1);
    int width = exponent.bitLength();
    for (int i = 0; i < width; i++) {
        s = s.multiply(s).mod(modulus);
        if(exponent.testBit(width-i-1))
            s = s.multiply(base).mod(modulus);
        else
            s.multiply(base).mod(modulus);
    }
    return s;
}
```

Listing 2.6: Safe example of Listing 2.5 [2]

2.1.3 Mixed Vulnerabilities

Some methods might suffer from both early-exit and control-flow based timing side-channel vulnerabilities. When this happens, the method is said to have a mixed timing side-channel vulnerability. To correct the vulnerability it is necessary to correct the early-exit part of the vulnerability and the control-flow part. This can be done in different ways. First, it can be corrected at the same time, meaning, that while analysing the method, the repair program corrects an early-exit or control-flow based timing side-channel vulnerability as they happen. Another option is to first correct one of the types of vulnerability and then correct the other type on the corrected version of the first type. For instance, we can first correct an early-exit timing side-channel vulnerability and then correct the control-flow based timing side-channel vulnerability in the previously repaired version.

As an example, Listing 2.7 shows a method with a mixed timing side-channel vulnerability (taken from the dataset of DifFuzz [1]) where the parameter “a” is a secret. Following the correction of an early-exit timing side-channel and then of a control-flow based timing side-channel, the final method should be similar to the example in Listing 2.8.

```
public static boolean sanity_unsafe(int a, int b) {
    int i = b;
    int j = b;
    if (b<0)
        return false;

    if (a<0) {
        return true;
    } else {
        while (i > 0) {
            i--;
        }
    }
    return false;
}
```

Listing 2.7: Example of a Mixed timing side-channel vulnerability in a method

```java
public static boolean sanity_safe(int a, int b) {
    System.out.println("a: " + a + ";b: " + b);
    int i = b;
    int j = b;
    if (b<0)
        return false;

    if (a>0) {
        while (i > 0) {
            i--;
        }
    } else {
        while (j > 0) {
            j--;
        }

    }
    return false;
}
```

Listing 2.8: Safe example of Listing 2.7

2.2 Automated Detection of Timing Side-Channel Vulnerabilities

Automatic detection of timing side-channel vulnerabilities has received substantial attention in recent years. This area has been pushed in several directions, which can be divided into two different paths, static and dynamic detection of timing side-channel vulnerabilities.

In 2017, Timos Antonopoulos et al. [2] developed a new way to prove the absence of timing side-channels. They attempted to use decomposition instead of self-composition to prove the absence of timing side-channels. Their idea as they put it was "... to partition the program's execution traces in such a way that each partition component is checked for timing attack resilience by a time complexity analysis and that per-component resilience implies the resilience of the whole program.". This means that their approach divides the program's execution traces into smaller and less complex partitions. Then each partition has their resilience to timing side-channels attacks checked through a time complexity analysis. The authors' idea is that the resilience of each component proves the resilience of the whole program. To ensure that any pair of program traces with the same public input has a component containing both traces, the construction of the partition is done by splitting the program traces at secret-independent branches. The authors' approach follows the demand-driven partitioning strategy that uses a regex-like notion that they call *trails*, which identifies sets of execution traces, particularly those influenced by tainted (or secret) data.

One extremely important concept discussed by the authors is the notion of *k-safety property*, which is a property that involves *k* terminating runs of a program.

Since in this paper, the authors want to prove the absence of timing side-channel vulnerabilities, they need *k* to be equal to 2, meaning 2 runs. So, any pair of executions whose computation uses the same public inputs should have similar running times, with a maximum difference based on the machine they run on. Although the authors focused on *k* equal to 2, their tool can be used for any value of *k*, greater than 2.

Before this paper, other approaches employed self-composition, where they aim to reason about multiple executions at once. However, the authors present the novel idea of decomposition, where they prove a non-relational property about a trace, instead of proving a relational property about every pair of execution traces. With that, the authors developed a tool called Blazer. That tool will first try to prove that a partition has tight bounds on running times, which means that there is a limit to the partition running time. If that is not possible, it will try to synthesize possible attacks. If there is a difference in running time caused by secret values, then there is a possible attack. To check the effectiveness of the tool, the authors evaluated it with a collection of 25 benchmarks, consisting of 12 benchmarks hand-crafted by the authors, 7 from the literature and 6 fragments of the DARPA STAC challenge problems [15].

Taking as a basis the work developed by Timos Antonopoulos et al., in 2017, Jia Chen et al. [3] presented the notion of ϵ-bounded non-interference, a variation of Goguen and Meseguer's non-interference principle [16]. The authors define *ϵ-bounded non-interference* as "[...], given a program *P* and a 'tolerable' resource deviation ϵ, we would like to verify that the resource usage of *P* does not vary by more than ϵ no matter what the value of the secret.". The execution time of an application can be affected by sources external to the application. As such, a minimum difference in execution time should be expected and must be accepted. This minimum change is what the authors denoted as ϵ. To simplify, *ϵ-bounded non-interference* means that regardless of the secret, the execution time of an application will not vary by more than ϵ. Given that ϵ-bounded non-interference is an instance of a *2-safety property*, this verification is known to be harder than standard safety properties and the former has not been as well-studied as the latter. To solve those challenges the authors combined "[...] relatively lightweight static taint analysis with more precise relational verification techniques for reasoning about *k-safety* (i.e., properties that concern interactions between *k* program runs).".

To verify the ϵ-bounded non-interference property the authors present a new program logic called *Quantitative Cartesian Hoare Logic (QCHL)*, which is at the core of their technique. With Quantitative Cartesian Hoare Logic the authors can "[...] prove triples of the form $\langle \phi \rangle$ S $\langle \psi \rangle$, where *S* is a program fragment and ϕ, ψ are first-order formulas that relate the program's resource usage (e.g., execution time) between an arbitrary pair of program runs.". Thanks to the QCHL logic, symbolically executing two copies of the program in lockstep allows a relational verification.

With that, the authors developed a tool called Themis. To prove that Themis works as a tool and compares favourably to Blazer both in terms of accuracy and running time, the authors executed Themis on several real-world Java applications, on the same examples used to test Blazer, on one benchmark from the DARPA STAC project [15], and on seven other benchmarks with known vulnerabilities collected from GitHub. With these tests, the authors showed that Themis can find previously unknown vulnerabilities in widely used Java programs, that it can confirm that the repaired versions of vulnerable programs do not exhibit the original vulnerability and that it compares favourably against Blazer.

These two papers improve the field of detection of side-channel vulnerabilities. However, both papers used static analysis. As such, in 2019, Shirin Nilizadeh et al. [1] decided to take a new approach to the field using dynamic analysis, introducing a new tool called DifFuzz. According to the authors, this tool "[...] uses a form of differential fuzzing[2] to automatically find program inputs that reveal side channels related to a specified resource, such as time, consumed memory, or response size.".

As previously mentioned, non-interference is too strong of a property, with regards to side-channel, particularly in timing side-channel. As such, like in the previous articles, DifFuzz does not strictly check non-interference, instead will try to find a set of values to be used as inputs so that the difference in resource, in this case, time, between secret-dependent paths will be as large as possible.

DifFuzz can be used in programs written in any language. However, in their paper, the authors' solution targets Java. DIfFuzz instruments a program to record its coverage and resource consumption along the paths that are executed. As such, the inputs must maximize the code coverage. For that is used the fuzz testing tool *American Fuzzy Lop* (AFL) [17], which using genetic algorithms, will mutate the inputs using byte-level coverage. However, AFL only supports programs written in C, C++, or Objective C, and DifFuzz is written in Java. To connect these two tools, the authors used Kelinci [18] that provides AFL-style instrumentation for Java programs. This way, AFL does not know about the Java program being tested. Then, the user must create a Fuzzing Driver, that parses the input provided by AFL and executes two copies of the code, measuring the cost difference between the two. That cost difference will be used to guide the AFL in the generation of more input values so the difference can be increased.

[2]Fuzzing is an automated testing technique in which invalid, unexpected or random data is provided as input to the program in test.

And this process is repeated for a predetermined time or until the user cancels the execution of the tool. A diagram of the tool functionality can be seen in Figure 2.1.

To assess the correctness of DifFuzz, the authors decided to apply it to widely-used Java applications and they found previously unknown vulnerabilities (later confirmed by the developers). They also applied DifFuzz to complex examples from the DARPA STAC [15] program. Additionally, they compared their tool with Blazer and Themis. Unlike Blazer and Themis, that can give false alarms given that both perform static analysis, DifFuzz will not give false alarms since it performs dynamic analysis; however, it can not prove the absence of vulnerabilities. To accurately compare DifFuzz with Blazer and Themis, the authors evaluated DifFuzz on the same benchmarks used for Blazer and Themis, and on their corrected versions. DifFuzz was able to find the same vulnerabilities as the other tools and also found vulnerabilities on corrected versions of the benchmarks of Themis and Blazer.

As the authors state, they achieved their goals. However, there is still more that can be done to improve this tool. Two of their proposed improvements are to add statistical guarantees to the tool and to possibly add automated repair methods to eliminate the vulnerabilities discovered by DifFuzz. The purpose of this project is to contribute to the latter

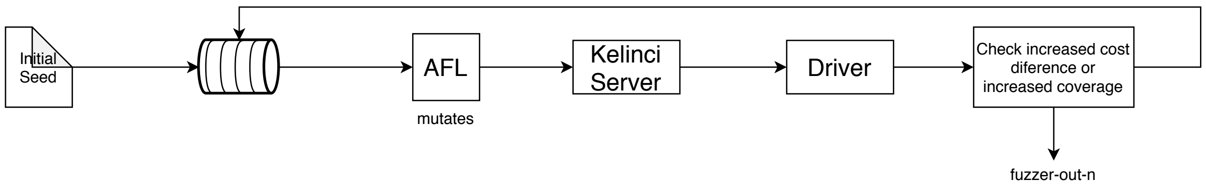

Figure 2.1: Overview of DifFuzz

2.3 Automated Repair of Software Bugs

The correction of a bug or vulnerability can be arduous and time-consuming because a correction can be hard to understand and/or hard to achieve. As such, the field of research in Automatic Repair is very active.

Automatic Repair consists in the correction of a software bug and/or vulnerability without human intervention. When a developer runs an automatic repair tool on an application, it should correct all the bugs in that application. This type of tool is increasingly more important given the growth of developed software and the greater emphasis on the use of Continuous Integration. As such, it is important that when developers check their software, using automatic detection tools, they can as easily automatically correct the bugs found.

There are several benefits of automatic repair. One is to use automatic repair to repair bugs during the development of an application. As previously mentioned, Continuous Integration is increasingly common in today's software development process. As such, automatic repair tools can be integrated into existing Continuous Integration pipelines. Automatic repair tools can also be used to repair vulnerabilities. Extremely popular libraries and frameworks are constantly checked for vulnerabilities, as such, automatic repair tools could be used to repair the vulnerabilities detected without requiring any human interaction, increasing the safety of software as a whole. As software and programming become more popular, increases the need for new and better ways to teach programming principles to newcomers. Automatic repair tools could be used to correct the code students create and guide them to better coding practices. More examples can be found in the work developed by Claire Le Goues et al. [19].

Automatic repair can be divided into two different techniques, behavioural repair and state repair. Behavioural repair as the name indicates consists in the modification of the program code, either the source or the binary code. This modification can happen in the IDE, a continuous integration server or even on the deployed software. This means that, once the automatic repair tool finishes, the developer can look at the code, and learn from it, seeing how certain parts of the code should have been done. State repair consists in the modification at runtime of the execution state. Being done at runtime means that there is no correction of the code, and as such, next time the application is run, it will have the same problems. Both families of automatic repair can be divided into several approaches. Given that in this project it will be used behavioural repair, a few of its approaches are covered in this report.

The approaches to behavioural repair can be split into different types that explain how the repair is processed, and what type of oracle[3] is used. The first approach is **Repair & Oracles**. Here the automatic repair is done concerning an oracle. The oracle used can be of varied types and its type influences how the repair is done. The oracle can be a *Test Suite*, which is an input-output specification. In this type of repair, a failing test case acts as a bug oracle, while the passing test cases are a regression oracle. This means that the failing test case identifies the bug, while the passing test cases identify all the functionality that should be maintained. There are tools developed with this type of repair such as Genprog [20] and Nopol [21]. The oracle can take into account the *pre and post conditions* of the application. The repair can also be driven by an *Abstract Behavioural Model*, such as a state machine encoding the object state and the corresponding method calls.

Another approach is called **Static Analysis**. Here the automatic repair tool can be built on top or be included in static analysis tools. In this case, it is part of the responsibility of a single tool the whole process of discovering a bug and correcting it. In that case, the user does not need to worry about what type of information should be given to the automatic repair tool. In this approach, the oracle is the static analysis itself.

The final approach is called **Crashing inputs**. Crashing input can sometimes be confused with the test suite, however, the main difference is that in the test suite, there are passing and failing test cases,

[3]Oracle indicates if the program works successfully, given the program and a way to detect if the program execution is as expected.

while crashing input simply “... refers to a violation of the non-functional contract ‘the program shall not crash”’ [22].

There are many more repair approaches, however, given that they do not belong to the scope of this work they will not be mentioned. However, more information can be found in the work developed by Martin Monperrus [22]. In this work, the focus is on automatic repair with the use of Test Suites.

2.3.1 Automated Repair Tools

In 2012, Claire Le Goues et al. [20] presented a generic method for automatic software repair called GenProg. They aimed to develop a way to repair applications without the need for formal specifications or program annotations. That way, their tool could repair an application without preparing the source code. Achieving this would be an outstanding accomplishment, since any developer could use GenProg to repair software that they were using, without needing to understand or modify the code. To accomplish this, GenProg receives as input the defected source code and a set of test cases. In the set of test cases, at least one of them **must** be a failing negative test case and a set of passing positive test cases. The negative test case encodes the fault to be repaired, meaning that it should be a use case where the bug or vulnerability to be corrected can be noticed. The set of positive test case encodes the set of functionality that can not be lost while repairing the bug. For instance, imagine an API where the bug to correct is a POST request with a specific message, which is encoded in the negative test case. If the set of positive test cases do not test the correct functionality of POST requests, then a simple repair for the bug is to simply remove the POST requests functionality. As such, when using GenProg it is of the utmost importance to make the test cases as complete as possible to avoid losing any functionality in the application while assuring the best possibility that the bug will be repaired. This means that it is vital to have great code coverage. GenProg uses *genetic programming* to search for a variant of the program that retains all required functionality but does not have the bug in question. Genetic programming as the name indicates is inspired by biological evolution and creates new program variations, called variants, using crossover and computational mutation. A fitness function evaluates each variant’s desirability, and the test cases are used to evaluate the desirability. The variants with high desirability are passed to the next generation so that the process can be repeated until a variant passes all test cases or a predetermined resource limit is reached. When developing GenProg, the authors decided that it should only use statements from the program to be repaired instead of inventing new code, because, according to the authors, the developer coded a correct version of the bug somewhere in the application.

To evaluate GenProg, the authors repaired 16 C programs, and on average, GenProg found a repair in 357 seconds. They also found that 77% of the trials produced a repair. In the 16 patches achieved, seven inserted code, seven deleted code, while two did both. One problem of GenProg is that it must be manually initialized. To reduce this manual initialization, the authors proposed a prototype of a closed-loop repair system where the repair process is launched by real-time detection of bugs, and the system input is recorded so that it can construct a negative test case automatically. To test the closed-loop, the authors focused the experiments on three benchmarks consisting of security vulnerabilities in servers. After running those experiments, the authors found that the GenProg did not break legitimate repairs and addressed the given errors. Given that the repair time is low, it does not negatively influence performance.

Although the authors achieve their goals, GenProg has some limitations. Such as the fact that GenProg relies on test cases to identify important functionality of the application and to identify the error to repair. This is a limitation because it is extremely important that the tests cases provided with the application have 100% code coverage, or at the very least as close to 100% as possible. Since GenProg will use the test cases to identify the functionality of the application and the bug to correct, any functionality that is not included in the test cases might be deleted. Also, some properties of an application can be hard to encode with test cases, such as race conditions, which GenProg cannot repair. Another problem is the fact that if the patched program passes the test suite provided as input, GenProg considers the repair acceptable. As such, the quality of the repair can be impacted by the size and scope of the test suite. Given that for the repair to work, the bug must already be known, this is a problem for the automation aspect of GenProg. Even though the authors' closed-loop system automatically detects bugs, it does not work for all types of faults. The final problem that the authors refer, is the fact that the experimental setup and parameters were chosen based on the performance, and as such might not be the optimal set of parameters values. GenProg was used to repair programs written in C, but the ideas and findings of the authors can be adapted to other languages and can be used as a starting point to new approaches, and as such, this paper is important regardless of the target programming language.

In 2017 Jifeng Xuan et al. [21] presented Nopol, a new approach to automatically repair buggy conditional statements. This approach takes as input a program and a set of test cases and outputs a patch for the inputted program with a conditional expression. The set of test cases passed as input must include the passing test cases to encode the expected behaviour of the application, and should include at least one failing test case that encodes the bug. Unlike GenProg, which follows a generic approach for automatic software repair, Nopol was built to focus on a specific type of bugs, bugs in conditional statements. More precisely, buggy if conditions and missing precondition bugs. Buggy if conditions occur when a bug is the condition of an 'if' statement. Missing precondition bugs happen when there should be a condition before a statement, such as detecting null pointer or an invalid index to access an array.

Nopol uses Ochiai, a spectrum-based ranking metric, as "[...] a fault localization technique to rank statements according to their suspiciousness of containing bugs.". The application' statements are ranked in a descending order based on their suspiciousness score. As such the statements with a higher suspiciousness score, closer to 1, will be at the top of the rank while the statements with the lowest suspiciousness score, closer to 0, will be at the bottom. The suspiciousness score indicates the likelihood that a statement contains a bug. If the statement is a condition, an 'if' statement, then Nopol considers it a buggy 'if' condition. If the statement is any other statement except a branch or loop statement, Nopol considers it a missing precondition candidate. Then each one of these statements is processed in a phase called *angelic fix localization*, where conditional values in 'if' statements are replaced by values that pass the failing test cases provided as input. If a value passes the failing test cases, it is called an *angelic value*. In the case of a non-loop and non-branch statement, the angelic fix localization skips that statement. If skipping the execution of that statement a failing test case passes, then a potential fix location has been found, meaning there is a possibility that there is a missing condition before that statement. In this work, the authors only consider missing precondition for single statements and not blocks. If an 'if' condition is used multiple times in a failing test case, there might exist a sequence of angelic values that pass that test. However, to avoid a combinatorial explosion, Nopol does not consider this situation. All the statements where Nopol finds an angelic value are identified as a *fix location*. Then, Nopol enters in the *runtime trace collection* phase. Here all test cases passed as input are run, to collect the execution context of the fix location. The execution context includes the expected outcome of conditional values, the static values in the program, and given that Nopol support automatic repair for object-oriented programs, it also collects the nullness of all variables and the output of state query methods[4]. As such, a repair created by Nopol might contain polymorphic calls. Next comes the *patch synthesis* phase, where the collected context is converted into a Satisfiability Modulo Theory (SMT) formula. If the SMT formula is satisfied, then that implies a program expression that preserves the behaviour while fixing the bug, that is, the expression passes all the test cases. The expression that satisfies the SMT formula is converted to source code. Otherwise, Nopol passes to the next statement in the ranking.

To evaluate Nopol, the authors executed it on a dataset with 22 real-world bugs from Apache Commons Math and Apache Commons Lang. In the dataset, there are 16 bugs with buggy if conditions and 6 bugs with missing preconditions. To build this dataset the authors extracted commits that modified a maximum of five files, where an if condition was modified or added. Out of the 22 bugs, Nopol only failed the repair of 5 bugs, four of which are related to timeout. From the 17 repairs, 13 are as correct as of the manual patches. The authors reported that the average repair time of one bug was 24.8 seconds.

However, as a new approach to automatic repair, Nopol has some limitations identified by the authors. When a failing test case executes both branches of an if condition, then Nopol can not find an angelic value. To solve this, the test cases should be refactored into smaller test cases, where only one branch

[4]A method without arguments and with a primitive return type

of the if condition is covered. Another limitation is an infinite loop caused by the angelic fix localization. To correct this, the authors suggest setting a maximum execution time, however as they point out this is impractical since it is hard to determine maximum execution time. Sometimes, in a given statement the values true and false can make a test case pass, and as such the synthesis might not work for missing preconditions. As it stands, Nopol only support calls to methods without parameters. As such, if the necessary patch needs to have in a condition a method with parameters, Nopol cannot provide it. The final limitation presented by the authors happens when in at least one test case, an object is null, and the patch to be created contains a call to state query method of that object.

Despite these limitations, Nopol can be considered a success and can be used as a tool in real work or it can be used as a foundation to new and improved approaches.

This paper is, in fact, an extension of previous work done by the same authors in 2014 [23]. However, it was chosen to discuss this paper because it adds an evaluation on a real-world bug dataset, so that it can be tested in an environment closer to reality and four detailed case studies that reference 4 bugs corrected by Nopol, where the comparison of the automatic repair to a manual-repair is always different.

2.4 Automated Repair of Timing Side-Channel Vulnerabilities

The correction of a timing side-channel vulnerability is an arduous and error-prone task. To reduce that hardship, there has been an effort to find ways to correct a timing side-channel vulnerability automatically. One of that example was Meng Wu et al. [24] that proposed a method based on program analysis and transformation to eliminate timing side-channel vulnerabilities. According to the authors, their solution produces a transformed program functionally equivalent to the original program but without instruction and cache timing side-channels. They also claim that they ensure that the number of CPU cycles taken to execute any path is independent of the secret data, and the cache behaviour of memory accesses is independent of the secret data in terms of hits and misses. Their method is implemented in LLVM (Low-Level Virtual Machine) and tested using libraries with a total of 19,708 lines of C/C++ code.

In their work, the authors identify a different type of timing side-channel vulnerability, cache-related timing side-channel vulnerability which means that the memory subsystem may behave differently depending on the values of the secret variables.

The method the authors created uses static analysis to identify the set of variables whose values depend on the secret inputs. Then, to decide if those variables lead to timing side-channel vulnerabilities, they check if the variables affect unbalanced conditional jumps, for instruction timing side-channel, or accesses to memory blocks across multiple cache lines, for cache-related timing side-channel vulnerabilities. After this analysis, to mitigate the leaks, code transformation is performed to equalize the execution time.

Applying their methods, the authors created the tool SC-Eliminator, that takes as input the program as LLVM bit-code and a list of secret variables and outputs the transformed program. This tool starts by doing a series of static analysis to identify the sensitive variables and their associated timing leaks. Next, the tool performs two transformations, one to eliminate the difference in execution time caused by unbalanced conditional jumps and the other to eliminate the difference in the number of cache hits/misses during the accesses of look-up tables.

Testing the tool, the authors claim that it is scalable, efficient and effective in removing instruction and cache related timing side-channels. They also claim that the transformed program created by the tool is only slightly bigger and slower than the original code.

2.5 Automated Testing of Java Software

Given that the idea behind this project is the automatic repair of timing side-channel vulnerabilities, meaning that the source code will be modified without human intervention, it is necessary to ensure that those modifications do not break any functionality of the modified code. For that, the original code must have unit tests that can then run on the repaired version of the code.

However, the datasets of vulnerable code that are used to test this project (e.g. DifFuzz's dataset) do not contain any unit tests. As such, before proposing a correction for any of the examples in those datasets, it is necessary to create unit tests for those examples. The problem is that creating unit tests is a time-consuming activity. Besides that, the creation of unit tests is prone to errors. For those reasons, automated approaches are of utmost importance.

The solution that this project adopts is the tool EvoSuite [25]. This tool automatically generates test cases with assertions for classes written in Java code. To do this, Evosuite uses an approach that generates and optimizes test suites to satisfy a coverage criterion. The generated tests use JUnit4. Evosuite can either create new test suites for the classes in the project or it can add new test cases to existing test suites. This tool can either be used on the command line or as a plugin for Eclipse, Intellij IDEA or maven. Once the tests are created, they are no different than tests created manually, meaning they can run like any other JUnit test. The only difference in the tests is that tests generated by EvoSuite rely on the EvoSuite framework and as such needs to be on the classpath.

3

System Architecture and Methodology

Contents

3.1 System Overview 27
3.2 Identification of Vulnerable Methods 28
3.3 Correction of Vulnerabilities 29

This chapter presents the architecture of *DifFuzzAR* and describes the methodology used to identify and repair timing side-channel vulnerabilities.

3.1 System Overview

DifFuzzAR is designed to work in conjunction with *DifFuzz*. In terms of its workflow, the tool needs to first identify the vulnerable method to be repaired. For this, the tool assumes the existence of a Driver file that can be used with *DifFuzz*. Once the vulnerable method is identified using the Driver, the tool will attempt to repair the method. In the current version of the tool, it will attempt to repair Early-Exit Timing Side-Channel vulnerabilities and Control-Flow Based Timing Side-Channel vulnerabilities.

DifFuzzAR was designed to be as modular as possible. This way if someone wants to add functionality to repair other types of vulnerabilities, they simply have to create a new independent module with all the code capable of repairing that vulnerability and add a call to the new module in the tool. The one thing that is intrinsic to the tool is the analysis of the Driver to identify the vulnerable method and the class it belongs to. Once this identification is done, the tool searches for that method and sends it to the module responsible for correcting an early-exit timing side-channel vulnerability. That module then creates a repaired version of the method and that version of the method is sent to the module responsible for correcting a control-flow based timing side-channel vulnerability. That module then creates another repaired version of the method and, given that it is the final module, the tool saves that method in a new file that is a copy of the original file. In case the user decides to add a new module to correct another vulnerability, can also decide the order by which each module repair the original code. However, the user needs to be aware that correction of vulnerabilities can add or exacerbate another vulnerabilities. As such, it is important to consider what is the order of the modules. For instance, the module responsible for the correction of an early-exit timing side-channel vulnerability can add or exacerbate a control-flow based timing side-channel, as such the module to correct an early-exit timing side-channel vulnerability needs to happen before the module to correct a control-flow timing side-channel vulnerability. Besides that, when creating a new module, the user must ensure that in some way the execution of the method produces a Spoon method representation, CtMethod, of the correct method. This can either be as a return value of the module, or the modification of the object the module receives as an argument. It will be that CtMethod, that the tool will write in a new file as its output.

A basic overview of the tool code can be seen in Algorithm 3.1. An overview of the architecture of the tool is also available in Figure 3.1.

Algorithm 3.1: Tool Code Overview

```
processDriver(driverPath);
modifyCode(driverPath);
write new file with modified method;
```

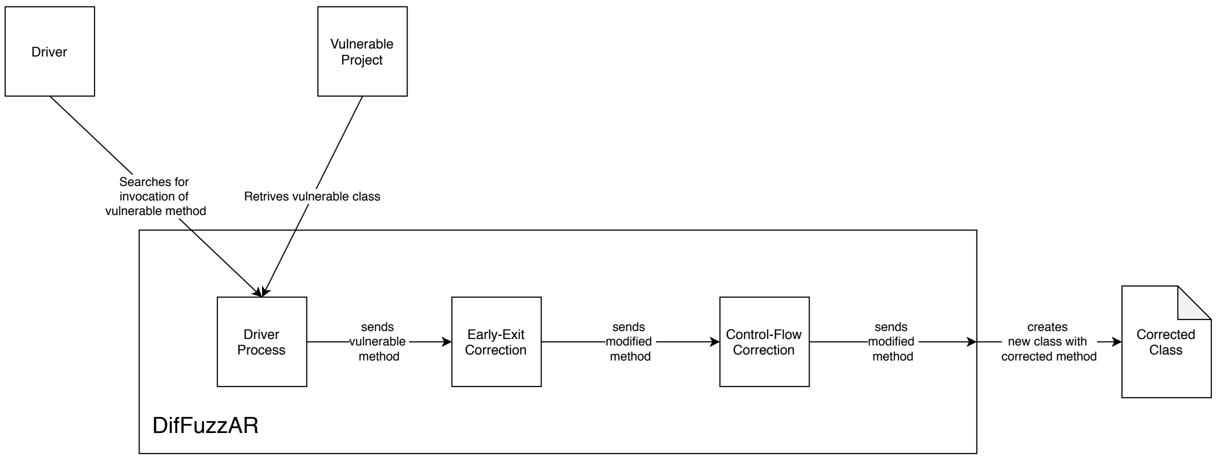

Figure 3.1: Overview of DifFuzzAR

3.2 Identification of Vulnerable Methods

The first task of the tool is to uncover the vulnerable method that is to be repaired. Since this tool should be used in conjunction with *DifFuzz*, the driver used for *DifFuzz* is also used to identify the vulnerable method. Doing this reduces the manual work of the users and reduces errors. However, this also means that the driver must be properly created so that the correct method is retrieved. For the tool to properly identify the vulnerable method, the method invocation should be immediately preceded by a call to Mem.clear(). This requirement follows from what is already stipulated for *DifFuzz*: while creating a driver for *DifFuzz*, it is required to call the method Mem.clear(). Therefore, to use a *DifFuzz* driver with *DifFuzzAR* it is simply a matter of ensuring that the vulnerable method is invoked immediately after the call to Mem.clear(). The only instructions that can be between the invocation of Mem.clear() and that of the vulnerable method are a constructor invocation, a 'try' keyword (meaning that the vulnerable method invocation is surrounded by a 'try' block), or an 'if' keyword (meaning that the vulnerable method invocation is the first instruction of the 'then' or 'else' block).

The search for the instruction after the Mem.clear() instruction is done twice since in the driver the vulnerable method will be invoked twice and that way the tool can discover what parameter of the method is the secret since it will be the one where a different argument is used in the two invocations of the method. As such, in the driver, the user must ensure to use the same argument for the public parameters and different ones for the secret. Taken as an example the two method invocations shown in Listing 3.1, the second parameter is considered by the tool as the secret, since it is the only argument that changes.

```
vulnMethod(a, b, c);

vulnMethod(a, d, c);
```

Listing 3.1: Vulnerable method invocation example

In the identification of the vulnerable method, the tool also finds the path to the class file where the method definition is, even if that class is an inner class in some package. The tool also validates its findings of the vulnerable method by comparing the two instances and checking if name, class, return type and the number of parameters are the same, while at least one argument is different. A basic overview of the process of identifying the vulnerable method from the driver can be seen in the Algorithm 3.2. In this process, given the path to the Driver file, the tool searches for the first occurrence of Mem.Clear(). Once it finds it, analyses the next instruction, which should be a method invocation. This method is considered by the tool as the vulnerable method to be corrected. The tool then searches for the second occurrence of the Mem.Clear() and repeats the process. With the two invocations of the vulnerable method, the tool will compare them to see if it is the same invocation, the arguments that differ in the two invocations are considered the secrets of the method.

Algorithm 3.2: Identification of vulnerable method using *DifFuzz* driver

```
1: f ← findDriverFile(driverPath)
2: instMem1, f' ← findMemClear(f)
3: vulnOpt1 ← recordNextInstruction(instMem1)
4: instMem2 ← findMemClear(f')
5: vulnOpt2 ← recordNextInstruction(instMem2)
6: valid ← compareInstructions(vulnOpt1, vulnOpt2)
```

3.3 Correction of Vulnerabilities

In the current version of *DifFuzzAR*, the correction of a vulnerability is done in two separate phases: the correction of an early-exit timing side-channel vulnerability followed by the correction of a control-flow based timing side-channel vulnerability. This way, there are two separate modules, each responsible for the correction of one type of timing side-channel vulnerability. As mentioned above, the addition of the correction of a new type of side-channel vulnerability is as simple as writing the code responsible for that correction and adding the module to the tool, as well as its invocation.

From the previous identification step, the tool knows which method was identified as vulnerable by *DifFuzz*. However, it does not know the specific instruction or set of instructions that cause the vulnerability. As such, the tool has to analyze the code and produce a correction that consists in a modification of the code to make its execution time as independent of the secret as possible. Algorithm 3.3 shows a basic overview of the correction process. If the vulnerable method has more than one return statement, then the tool considers it to have an early-exit and so the tool starts by correcting that vulnerability. After that is done, the tool executes the module responsible for the correction of control-flow based timing side-channel vulnerabilities.

Algorithm 3.3: Overview of the repair process

```
1: if numberReturns >1 then
2:     vulnMethod ← repairEarlyExit(vulnMethod)
3: end if
4: vulnMethod ← repairControlFlow(vulnMethod)
```

3.3.1 Correction of Early-Exit Timing Side-Channel Vulnerabilities

Early-Exit timing side-channel vulnerabilities occur when there is an exit from the method that depends on the secret and the execution time of the method is significant after that exit. The correction of this type of timing side-channel vulnerability consists of the elimination of all ‘return’ statements except the last one. However, the result of the execution of the method should be the same after the modification. For that reason, every ‘return’ statement of the method will be replaced with an assignment of the value being returned to a variable. That variable will either be the variable returned in the final return (if it returns a variable) or a new one created with the return type of the method.

A more detailed description of the correction of this type of vulnerability is available in Section 4.3. In Algorithm 3.4 is shown an overview of the correction process for early-exit timing side-channel vulnerability. The tool starts by obtaining the element returned in the final return of the method. This element should be a variable, if it is not the tool creates a new variable, were its type is the return type of the method, and is initialized with the element obtained, referred from now on as return variable. Then, the tool analysis every instruction of the method in search of a return statement, which will be replaced by an assignment to the return variable with the value being returned. If that return statement happens after a ‘while’ block, then the instruction is added before the ‘while’ block. If it is the last return statement, then the value being returned is altered to be the return variable. If the return statement is inside an ‘if’ statement, then the condition of the ‘if’ statement is saved to be used to protect the variables used in the condition. If the instruction under analysis uses any variable saved to be protected, then that statement will be inside the ‘then’ block of a new ‘if’ statement, where the condition is the combination of the negation of every condition that variable was part of. In the end is created a new method, with the early-exit vulnerability corrected.

This simple modification can create or exacerbate a control-flow based timing side-channel vulnerability. For instance, if an early-exit happens inside a cycle, where the stopping condition depends on a

Algorithm 3.4: Correction of Early-Exit Timing Side-Channel Vulnerabilities

```
1: returnElem ← obtainElemReturnedMethod(vulnMethod)
2: if !isVariable(returnElemen) then
3:    returnElem ← createVariable(returnElem)
4: end if
5: instruction ← getNextInstruction(vulnMethod)
6: while exist(instruction) do
7:    if isReturn(instruction) then
8:       if afterWhileBlock(instruction) then
9:          addAssignmentBeforeWhile(instruction, returnElem)
10:      else if lastReturn(instruction) then
11:         changeReturnElem(instruction, returnElem)
12:      else
13:         replaceWithAssignment(instruction, returnElem)
14:         if insideIf(instruction) then
15:            condition ← saveCondition(instruction)
16:         end if
17:      end if
18:   else if isVariableProtected(instruction) then
19:      addIfToVariable(instruction)
20:   end if
21:   instruction ← getNextInstruction(vulnMethod)
22: end while
23: newMethod ← saveChanges()
```

secret, then the effect of the existing control-flow based timing side-channel vulnerability becomes more prominent, i.e., the difference in execution time depending on the secret is greater since it will have more instructions to execute. For instance, take as an example the listing 3.2, where the parameter “pw” is a secret, from the application **example_PWCheck**, from *DifFuzz* dataset. Here, the original method, where the secret is the argument *pw*, suffers from a mixed timing side-channel vulnerability. There are multiple return statements, indicating an early-exit timing side-channel vulnerability and, the conditions of the ‘if’ branches depend on the secret indicating a control-flow based timing side-channel vulnerability. However, after correcting the early-exit timing side-channel vulnerability, seen in the listing 3.3, the effects of the control-flow based timing side-channel become more prominent, mostly because there are instructions dependent on the secret that happen repeatedly, in this case, the ‘if’ statement inside the cycle.

```java
public static boolean pwcheck1_unsafe(byte[] guess, byte[] pw) {
    if (guess.length != pw.length) {
        return false;
    }
    int i;
    for (i = 0; i < guess.length; i++) {
        if (guess[i] != pw[i]) {
            return false;
        }
    }
    return true;
}
```

Listing 3.2: Original Vulnerable Method

```
public static boolean pwcheck1_unsafe$Modification(byte[] guess, byte[] pw) {
    boolean $1 = true;
    if (guess.length != pw.length) {
        $1 = false;
    }
    int i;
    for (i = 0; i < guess.length; i++) {
        if (((i < guess.length) && (i < pw.length)) && (guess[i] != pw[i])) {
            $1 = false;
        }
    }
    return $1;
}
```

Listing 3.3: Vulnerable Method after early-exit correction

3.3.2 Correct Control-Flow Timing Side-Channel Vulnerabilities

Control-flow based timing side-channel vulnerabilities happen when a slow operation is triggered by a condition involving the secret. This can be the invocation of a slow executing method if a given condition involving a secret is met or a cycle where the stopping condition depends on the secret. The correction of this type of vulnerability involves the modification of the stopping condition of cycles that depend on a secret to depend on a public argument or the replication of the block of instructions of the ‘then’ block to the ‘else’ block, and vice-versa, of an ‘if’ statement where the condition depends on the secret.

A more detailed explanation of the correction process of control-flow based timing side-channel vulnerabilities is provided in Section 4.4. Algorithms 3.5 and 3.6 show an overview of the correction process for this type of vulnerabilities. In this process, the tool starts by creating a list of the secrets and one of the public arguments. The list of public arguments is final, while the list of secrets is updated during the analysis of the method. Every time, a variable is assigned with a value dependent on a secret, that variable is added to the list of secrets. The tool also creates a map, to connect the newly created variables with the old variables being replaced. The tool then starts to analyse each instruction, taking actions according to the type of instruction and where the instruction happens.

If the instruction found is an assignment and that needs to be modified, then a new variable is created and it is added to the list of replacement with the existing variable, and the instruction is changed so that the variable being assigned to is the newly created one.

If the instruction found is a ‘for’ statement and the stopping condition uses a secret, the tool will attempt to change the condition to use a public argument instead of the secret. This public argument must be of the same type as the secret in the stopping condition. When the tool finds a ‘for’ statement it will retrieve the body of the ‘for’ and will analyse each instruction of that block.

If the instruction found is an ‘if’ statement then the tool will retrieve the ‘then’ and ‘else’ blocks. If the condition uses a secret, then the tool will try to modify the instructions of the ‘then’ block and then of the ‘else’ block, producing two new blocks with the modified versions of the instructions. Then, the modified version of the ‘then’ block is added to the ‘else’ block and the modified version of the ‘else’ block is added to the ‘then’ block. Otherwise, the tool will analyse each instruction of both blocks without adding new instructions to either block.

If the instruction is an invocation, the tool will retrieve the target of that invocation. If the target is a secret, then the tool will create a new variable to replace the target.

If the instruction is a local variable, the tool will retrieve the assigned value. If that value uses a secret, then the variable assigned to will be considered a secret. If the value being assigned does not use any variable that is used in the condition of the ‘if’ statement this instruction belongs to, then a new variable to replace the variable assigned to is created.

If the instruction is a loop statement, then the tool will retrieve its body and will analyse each instruction of the body.

If the instruction is an operator assignment, then the tool will create a new variable to replace the one being assigned to.

If the instruction is a ‘try’ block, then the tool will retrieve its body and will analyse its instructions.

If the instruction is a unary operator, the tool will retrieve the variable used. If that variable was already replaced, then the tool will obtain the variable created as a replacement and will replace the variable in the unary operator with the variable created for replacement.

If the instruction is a ‘while’ statement, the tool will replace the variables used in the stopping condition, either by variables already created as replacements or with newly created variables. Then the tool retrieves the body of the tool and will analyse its instructions.

In the end is created a new method, with the control-flow based timing side-channel vulnerability corrected.

Algorithm 3.5: Correction of Control-Flow Based Timing Side-Channel Vulnerabilities - PART 1

```
secrets ← createListOfSecrets(vulnMethod)
public ← createListOfPublic(vulnMethod)
replacements ← newMap()
instruction ← getNextInstruction(vulnMethod)
while exist(instruction) do
   if isAssignment(instruction) then
      if valueAssignedUsesSecret(instruction, secrets) then
         variable ← getVariableAssignedTo(instruction)
         secrets ← addToSecrets(variable, secrets)
      end if
      if toModify(instruction) then
         newVar ← createNewVariable(instruction)
         addToReplacements(replacements, instruction, newVar)
         changeVariableAssignedTo(instruction, newVar)
      end if
   else if isForStatement(instruction) then
      if conditionUsesSecret(instruction, secrets) then
         changeConditionToUsePublic(instruction, secrets, public)
      end if
      traverseForBody(instruction)
   else if isIfStatement(instruction) then
      thenBlock ← getThenBlock(instruction)
      elseBlock ← getElseBlock(instruction)
      if conditionUsesSecret(instruction, secrets) then
         modThen ← modifyInstructions(thenBlock)
         modElse ← modifyInstructions(elseBlock)
         addToStartOfBlock(modThen, thenBlock)
         addToStartOfBlock(modElse, elseBlock)
      else
         traverseBlock(thenBlock)
         traverseBlock(elseBlock)
      end if
   else if isInvocation(instruction) then
      target ← getInvocationTarget(instruction)
      if isSecret(target, secrets) then
         newTarget ← createNewVariable(target)
         addToReplacements(replacements, instruction, newTarget)
         replaceTarget(instruction, newTarget)
      end if
```

Algorithm 3.6: Correction of Control-Flow Based Timing Side-Channel Vulnerabilities - PART 2

```
40:   else if isLocalVariable(instruction) then
41:     assigned ← getValueAssigned(instruction)
42:     if usesSecret(assigned, secrets) then
43:       variableAssigned ← getVariableAssignedTo(instruction)
44:       secrets ← addtoSecrets(variableAssigned, secrets)
45:     else if !isPartOfConditionOfParentIf(instruction, assigned) then
46:       newVar ← createNewVariable(instruction)
47:       addToReplacements(replacements, instruction, newVar)
48:       changeVariableAssignedTo(instruction, newVar)
49:     end if
50:   else if isLoopStatement(instruction) then
51:     traverseLoopBody(instruction)
52:   else if isOperatorAssignment(instruction) then
53:     newVar ← createNewVariable(instruction)
54:     addToReplacements(replacements, instruction, newVar)
55:     changeVariableAssignedTo(instruction, newVar)
56:   else if isTryBlock(instruction) then
57:     traverseTryBody(instruction)
58:   else if isUnaryOperator(instruction) then
59:     var ← getVariable(instruction)
60:     if isInReplacements(replacements, var) then
61:       replacement ← getReplacement(replacements, var)
62:       changeVariableUsed(instruction, replacement)
63:     end if
64:   else if isWhileStatement(instruction) then
65:     condition ← getStoppingCondition(instruction)
66:     if usesReplacedVariable(condition, replacements) then
67:       condition ← getReplacement(replacements, instruction)
68:     else
69:       condition ← createNewVariable(condition)
70:       addToReplacements(replacements, instruction, condition)
71:     end if
72:     updateStoppingCondition(instruction, condition)
73:     traverseWhileBody(instruction)
74:   end if
75:   instruction ← getNextInstruction(vulnMethod)
76: end while
77: newMethod ← saveChanges()
```

4 Implementation

Contents

4.1 Software Tools Used 39
4.2 Driver Processor 40
4.3 Correction of Early-Exit Timing Side-Channel Vulnerabilities 53
4.4 Correction of Control-Flow Based Timing Side-Channel Vulnerabilities 56
4.5 Correction of Mixed Timing Side-Channel Vulnerabilities 60

This chapter presents the implementation details of *DifFuzzAR*. It starts by describing the development of the component that, based on a *DifFuzz* driver, identifies the vulnerable method: the driver processor. This is followed by a description of the two corrections currently implemented to fix early-exit and control flow timing side-channel vulnerabilities. We also discuss the case when a method has both types of vulnerabilities. It is also explained how the tool was developed so that it enables modularity. Finally, the tools used for refactoring and for ensuring that refactorings are correct are described.

4.1 Software Tools Used

4.1.1 Refactoring Tools

To create this it was necessary to find a way for the tool to be able to not only read the code but also write it. For this, it could be used any Java I/O library. The problem is that using any library would mean that when reading the code, the tool would not know what type of instruction it was reading. This means that that capability would have to be added to the tool. Which means, that not only was it necessary for the tool to know how to repair a timing side-channel vulnerability but also, to know what type of instruction it was reading, for instance, it needs to know the difference between an 'if' statement and a 'for' statement. That would make this tool, extremely complex and would deviate the focus from the correction of timing side-channel vulnerabilities to the ability to understand code.

For that reason, it was necessary to find a tool or library that would allow the tool, to read every instruction of a Java file and know what type of instruction it was reading. To that purpose, it was found two possibilities. The first is Javassist [26], Java Programming Assistant. This is a class library that allows the editing of bytecodes in Java and enables Java programs to define new classes at runtime and to modify a class file loaded by JVM. However, this tool is more focused on Java bytecode edition and it would require more control to know when the tool was reading, the instructions of a 'then' block of an 'if' statement.

The second tool found is, Spoon [27], which is a metaprogramming library to analyze and transform Java source code. One big advantage Spoon has over Javassist is the fact that it creates AST (Abstract Syntax Tree) of the code read. This means an instruction would know it belongs to the body of a cycle, and not directly to the method. As such, Spoon was chosen to be used in the tool. With that, the tool can read the Driver in search of a specific instruction and know when an instruction is inside a 'try' block for instance. By providing an AST of the code, it becomes easier to add instructions in a specific point of the method, like adding the modification of a 'then' block to the 'else' block.

4.1.2 Automated Test Generation

All the corrected examples were tested using the tool **EvoSuite** [25]. This tool provides a plugin that can be added as part of a maven project and used to generate tests. For that, it was created a maven project for each example that the correction was considered valid. In that project .pom file, EvoSuite was added as a dependency alongside the application dependencies. Then, the project with the original method, meaning the method before the correction of the tool, is built and tests are created by EvoSuite. Executing these tests, they should all pass. After that, the vulnerable method is replaced by the corrected method generated by the tool. The tests are now executed again, this time testing if the corrected method is semantically correct, which means that to the same inputs it produces the same outputs as the original method. DifFuzz can be executed on all examples considered semantically correct, to check if there is no vulnerability.

4.2 Driver Processor

To reduce the manual work of users, and since *DifFuzzAR* is designed to work in conjunction with *DifFuzz*, it was decided that the information about the vulnerable method would be retrieved directly from the driver used by *DifFuzz*. It is assumed that the driver points correctly to the vulnerable method and that it is part of the project folder containing the code to repair.

To identify the vulnerable method, the structure of available drivers in the dataset provided by *DifFuzz* was analyzed. It was observed that a correct driver should measure the execution time of an invocation of the vulnerable method. For that, it is necessary to invoke the method Mem.clear() right before the invocation of the vulnerable method, so that the current cost of execution is reset. As such, the tool checks each instruction present in the Driver and compares it to the instruction Mem.clear(). Once that instruction is found then it is assumed that the next instruction of the Driver will be the invocation of the vulnerable method. This can be seen in listing 4.1

```java
public static void main(String[] args) {
    if (args.length != 1) {
        System.out.println("Expects file name as parameter");
        return;
    }

    /* Read input. */
    char[] validPassword_userInputArray_public;
    String validPassword_public;
    String storedPassword_valid_secret1;
    char[] storedPassword_array_invalid_secret2;
    String storedPassword_invalid_secret2;

    PasswordEncryptor pe = new ClearTextPasswordEncryptor(SAFE_MODE);

    List<Character> values = new ArrayList<>();
    try (FileInputStream fis = new FileInputStream(args[0])) {
        int i = 0;
        int value;
        while (((value = fis.read()) != -1) && (i < MAX_PASSWORD_LENGTH)) {
            /* each char value must be between 0 and 127 and a printable
                character */
            value = value % 127;
            char charValue = (char) value;
            if (Character.isAlphabetic(charValue) || Character.isDigit(charValue)
                ) {
                values.add(charValue);
                i++;
            }
        }

        /* input must be non-empty */
        if (i == 0) {
            throw new RuntimeException("not enough data!");
        }

        validPassword_userInputArray_public = new char[values.size()];
        for (i = 0; i < values.size(); i++) {
            validPassword_userInputArray_public[i] = values.get(i);
        }
        validPassword_public = new String(validPassword_userInputArray_public);

        /* use a new String object to not have the same object, which might
            result in an early return during comparison */
        storedPassword_valid_secret1 = new String(pe.encrypt(validPassword_public
            ));

        /* ensure same length for secrets */
        storedPassword_array_invalid_secret2 = new char[
            storedPassword_valid_secret1.length()];

        i = 0;
        while (((value = fis.read()) != -1) && (i <
            storedPassword_array_invalid_secret2.length)) {
            /* each char value must be between 0 and 127 and a printable
                character */
            value = value % 127;
            char charValue = (char) value;
            if (Character.isAlphabetic(charValue) || Character.isDigit(charValue)
                ) {
                storedPassword_array_invalid_secret2[i] = charValue;
                i++;
            }
        }

        /* storedPassword_array_invalid_secret2 must be completed */
        if (i != storedPassword_array_invalid_secret2.length) {
            throw new RuntimeException("not enough data!");
        }

        storedPassword_invalid_secret2 = new String(
            storedPassword_array_invalid_secret2);

    } catch (IOException e) {
        System.err.println("Error reading input");
        e.printStackTrace();
        return;
    }

    System.out.println("public user input (password): " + validPassword_public);
    System.out.println("secret1 (valid stored password): " +
        storedPassword_valid_secret1);
    System.out.println("secret2 (invalid stored password): " +
        storedPassword_invalid_secret2);

    Mem.clear();
    boolean valid1 = pe.matches(validPassword_public,
        storedPassword_valid_secret1);
    long cost1 = Mem.instrCost;
    System.out.println("valid1=" + valid1);
    System.out.println("cost1=" + cost1);

    Mem.clear();
    boolean valid2 = pe.matches(validPassword_public,
        storedPassword_invalid_secret2);
    long cost2 = Mem.instrCost;
    System.out.println("valid2=" + valid2);
    System.out.println("cost2=" + cost2);

    long diff = Math.abs(cost1 - cost2);
    Kelinci.addCost(diff);
    System.out.println("diff=" + diff);

    System.out.println("Done.");
}
```

Listing 4.1: Example of simple Driver

After implementing this strategy, the tool was tested with the 58 drivers of all the examples provided with the *DifFuzz* dataset. It was observed that there were Drivers the tool was not capable of finding the invocation of the vulnerable method. After analysing the drivers where the tool could not find the correct invocation of the vulnerable method, it was found that some drivers took a slight deviation from the most common pattern. In particular, those drivers did not invoke the vulnerable method immediately after the invocation of Mem.clear(). After some deliberation, it was concluded that this slight deviation should be deemed valid and the tool was modified so that it could find the invocation of the vulnerable method in those examples. This deviation could be aggregated in three groups which simplified the modifications to the tool:

Group 1 The simplest deviation occurs when an object is created between the invocation of Mem.clear() and the invocation of the vulnerable method. This normally happens when the vulnerable method is an instance method and the object needed to invoke it is created before the invocation. This can be seen in listing 4.2.

```java
public static void main(String[] args) {
     if (args.length != 1) {
         System.out.println("Expects file name as parameter");
         return;
     }
     int n = 3;
     String public_ihash;
     String secret_iPassword1;
     String secret_iPassword2;

  // Read all inputs.
     List<Character> values = new ArrayList<>();
     try (FileInputStream fis = new FileInputStream(args[0])) {
         byte[] bytes = new byte[Character.BYTES];
         int i = 0;
         while (((fis.read(bytes)) != -1)) {
             char value = ByteBuffer.wrap(bytes).getChar();
             values.add(value);
             i++;
         }
     } catch (IOException e) {
         System.err.println("Error reading input");
         e.printStackTrace();
         return;
     }
     if (values.size() < n) {
         throw new RuntimeException("Too less data...");
     }

     int m = values.size() / n;
     System.out.println("m=" + m);

     // Read secret1.
     char[] secret1_arr = new char[m];
     for (int i = 0; i < m; i++) {
         secret1_arr[i] = values.get(i);
     }
     secret_iPassword1 = new String(secret1_arr);
```

```

        // Read secret2.
        char[] secret2_arr = new char[m];
        for (int i = 0; i < m; i++) {
            secret2_arr[i] = values.get(i + m);
        }
        secret_iPassword2= new String(secret2_arr);

        // Read public.
        char[] public_arr = new char[m];
        for (int i = 0; i < m; i++) {
            public_arr[i] = values.get(i + 2 * m);
        }
        public_ihash = new String(public_arr);

        System.out.println("secret_iPassword1=" + secret_iPassword1);
        System.out.println("secret_iPassword2=" + secret_iPassword2);
        System.out.println("public_ihash=" + public_ihash);

       Mem.clear();
       OSecurityManager manager = new OSecurityManager();
       manager.checkPassword_safe(secret_iPassword1, public_ihash);

       long cost1 = Mem.instrCost;
       System.out.println("cost1=" + cost1);

       Mem.clear();
       manager.checkPassword_safe(secret_iPassword2, public_ihash);

       long cost2 = Mem.instrCost;
       System.out.println("cost2=" + cost2);

       Kelinci.addCost(Math.abs(cost1 - cost2));
       System.out.println("|cost1-cost2|=" + Math.abs(cost2-cost1));

       System.out.println("Done.");
   }
```

Listing 4.2: Example of Driver with constructor after Mem.clear()

Group 2 A second deviation is when after the instruction Mem.clear() a 'try' block appears. When this happens, the vulnerable method is considered to be the first instruction of the 'try' block. This can be seen in listing 4.3.

```java
public static void main(String[] args) {
    if (args.length != 1) {
        System.out.println("Expects file name as parameter");
        return;
    }

    int startRow, rowsCount; // public
    int usersLength1, usersLength2; // secret

    /* Read input file. */
    try (FileInputStream fis = new FileInputStream(args[0])) {
        int value;

        if ((value = fis.read()) == -1) {
            throw new RuntimeException("Not enough data!");
        }
        startRow = Math.abs(value) % MAX_INT_VALUE;

        if ((value = fis.read()) == -1) {
            throw new RuntimeException("Not enough data!");
        }
        rowsCount = Math.abs(value) % MAX_INT_VALUE;

        if ((value = fis.read()) == -1) {
            throw new RuntimeException("Not enough data!");
        }
        usersLength1 = Math.abs(value) % MAX_INT_VALUE;

        if ((value = fis.read()) == -1) {
            throw new RuntimeException("Not enough data!");
        }
        usersLength2 = Math.abs(value) % MAX_INT_VALUE;

    } catch (IOException e) {
        System.err.println("Error reading input");
        e.printStackTrace();
        return;
    }

    System.out.println("startRow=" + startRow);
    System.out.println("rowsCount=" + rowsCount);
    System.out.println("usersLength1=" + usersLength1);
    System.out.println("usersLength2=" + usersLength2);

    UsersTableModelServiceImpl ser1 = new UsersTableModelServiceImpl(
        usersLength1);
    long cost1 = 0L;
    Mem.clear();
    try {
        // cost1 = Mem.instrCost;
        String[][] res1 = ser1.getRows(startRow, rowsCount, null, null, true);
        // cost1 = Mem.instrCost;
        cost1 = res1.length;
    } catch (IndexOutOfBoundsException e) {
        // ignore;
    }
    System.out.println("cost1=" + cost1);

    UsersTableModelServiceImpl ser2 = new UsersTableModelServiceImpl(
        usersLength2);
    long cost2 = 0L;
    int startRow2 = startRow2;
    int rowsCount2 = rowsCount;
    Mem.clear();
    try {
        String[][] res2 = ser2.getRows(startRow2, rowsCount2, null, null, true
            );
        // cost2 = Mem.instrCost;
        cost2 = res2.length;
    } catch (IndexOutOfBoundsException e) {
        // ignore;
    }
```

```
    System.out.println("cost2=" + cost2);

    long diffCost = Math.abs(cost1 - cost2);
    Kelinci.addCost(diffCost);
    System.out.println("diff=" + diffCost);
    System.out.println("Done.");

}
```

Listing 4.3: Example of Driver with try block after Mem.clear()

Group 3 The third deviation occurs when after the invocation of the instruction Mem.clear() an ‘if’ statement appears. When this happens, the invocation of the vulnerable method is considered to be the first statement of either the ‘then’ or ‘else’ block. This normally happens when the driver used for the safe and unsafe versions of an example are similar and the difference is only in the value assigned to a boolean variable. That variable will then be used as a condition of an ‘if’ to decide which method to invoke (either the safe or unsafe version). An example of this is:

```
public static void main(String[] args) {
    if (args.length != 1) {
        System.out.println("Expects file name as parameter");
        return;
    }

    boolean pwCorrect;
    String user = "";
    String pw = "";

    int minNumberOfBytes = Byte.BYTES + 2 * Character.BYTES;
    int maxNumberOfBytes = Byte.BYTES + USERNAME_MAX_LENGTH * Character.BYTES
            + PASSWORD_MAX_LENGTH * Character.BYTES;

    List<Byte> valueList = new ArrayList<>();

    try (FileInputStream fis = new FileInputStream(args[0])) {

        /* Read all bytes up to the specified maximum. */
        byte[] values = new byte[1];
        int i = 0;
        while (fis.read(values) != -1 && i < maxNumberOfBytes) {
            valueList.add(values[0]);
            i++;
        }
        if (i < minNumberOfBytes) {
            throw new RuntimeException("Not enough data!");
        }

        /* Determine boolean value for password correctness. */
        pwCorrect = valueList.get(0) > 0 ? true : false;

        /* Find available sizes for strings. */
        int remainingSize = valueList.size() - 1;
        int usernameSize;
        int pwSize;
        if ((remainingSize / 2) >= (USERNAME_MAX_LENGTH * Character.BYTES)) {
            usernameSize = USERNAME_MAX_LENGTH * Character.BYTES;
        } else {
            usernameSize = remainingSize / 2;
        }
        if (usernameSize % Character.BYTES == 1) {
            usernameSize--;
        }
        pwSize = Math.min(remainingSize - usernameSize, PASSWORD_MAX_LENGTH *
            Character.BYTES);
        if (pwSize % Character.BYTES == 1) {
            pwSize--;
        }

        /* Assign bytes to the remaining strings. */
        for (int j = 1; j < 1 + usernameSize; j += Character.BYTES) {
            byte[] bytes = new byte[Character.BYTES];
            for (int k = 0; k < Character.BYTES; k++) {
                bytes[k] = valueList.get(j + k);
            }
            user += ByteBuffer.wrap(bytes).getChar();
        }
        for (int j = 1 + usernameSize; j < 1 + usernameSize + pwSize; j +=
            Character.BYTES) {
            byte[] bytes = new byte[Character.BYTES];
            for (int k = 0; k < Character.BYTES; k++) {
                bytes[k] = valueList.get(j + k);
            }
            pw += ByteBuffer.wrap(bytes).getChar();
        }

        System.out.println("user=" + user);
        System.out.println("pw=" + pw);

    } catch (IOException e) {
        System.err.println("Error reading input");
        e.printStackTrace();
        return;
    }

    UsernamePasswordCredentials cred = new UsernamePasswordCredentials(user,
        pw, ""); // public info

    /* Create Connection to database. */
    DataSource ds = JdbcConnectionPool.create("jdbc:h2:~/pac4j-fuzz", "sa", ""
        );
    DBI dbi = new DBI(ds);

    DbAuthenticator dbAuth = new DbAuthenticator();
    dbAuth.dbi = dbi;

    /* Prepare database. */
    Handle h = dbi.open();
    try {
        String processedPW = dbAuth.getPasswordEncoder().encode(pw);
        if (!pwCorrect) {
            processedPW = processedPW.substring(0, processedPW.length() - 1)
                    + (char) (processedPW.charAt(processedPW.length() - 1) +
                        1);
        }
        h.execute("insert into users (id, username, password) values (1, ?, ?)
            ", user, processedPW); // add user
    } catch (Exception e) {
        throw new RuntimeException(e);
    } finally {
        h.close();
    }

    Mem.clear();
    boolean authenticated1 = false;
    try {
        if (RUN_UNSAFE_VERSION) {
            dbAuth.validate_unsafe(cred);
        } else {
            dbAuth.validate_safe(cred);
        }
        if (cred.getUserProfile() != null) {
            authenticated1 = true;
        }
    } catch (Exception e) {
    }
    long cost1 = Mem.instrCost;
    System.out.println("authenticated1: " + authenticated1);
    System.out.println("cost1=" + cost1);

    /* Prepare database. */
    h = dbi.open();
    try {
        h.execute("delete from users where id = 1"); // remove user
    } catch (Exception e) {
        throw new RuntimeException(e);
    } finally {
        h.close();
    }

    Mem.clear();
    boolean authenticated2 = false;
    try {
        if (RUN_UNSAFE_VERSION) {
            dbAuth.validate_unsafe(cred);
        } else {
            dbAuth.validate_safe(cred);
        }
        if (cred2.getUserProfile() != null) {
            authenticated2 = true;
        }
    } catch (Exception e) {
    }
    long cost2 = Mem.instrCost;
    System.out.println("authenticated2: " + authenticated2);
    System.out.println("cost2=" + cost2);

    Kelinci.addCost(Math.abs(cost1 - cost2));

    /* Clean database. */
    h = dbi.open();
    try {
        h.execute("delete from users");
    } catch (Exception e) {
        throw new RuntimeException(e);
    } finally {
        h.close();
    }

    System.out.println("Done.");
}
```

Listing 4.4: Example of Driver with if block after Mem.clear()

To resolve this case, it is necessary to record the variable and its value. When the tool finds the 'if' statement where its condition is the variable found, the value of the variable is used to decide whether to look in the first instruction of the 'then' block or the 'else' block.

Besides finding the name of the vulnerable method the tool also needs to retrieve the exact path for the file containing the vulnerable method so that it can correct it. To do this the tool retrieves the name of the class of the method. This can be done by checking the name of the class used to invoke the method when the method is static or it can check the type of the object used to invoke the method when the method is an instance method. When the tool finds the name of the class it can also retrieve the full package that class belongs to.

After these modifications, the tool was capable of finding the correct vulnerable method of all the examples in the *DifFuzz* dataset.

4.3 Correction of Early-Exit Timing Side-Channel Vulnerabilities

The main idea behind the repair of an early-exit timing side-channel vulnerability is to ensure that the method contains a single return. As such, the first thing to do is to check if the method needs repairing by checking if it contains at least two return statements.

Then, it is necessary to define what will be returned. The returned value will always be a variable whose value is what would be returned if no modifications were done. If the method already returns a variable, then that variable will be used as a replacement for the other returns of the method. Otherwise, a new variable will be created with the starting value equal to the final return. That variable will hereafter be referred to as the return variable.

Afterwards, each instruction of the method is analysed. Whenever the instruction under analysis is a return statement it is transformed into an assignment. The returned value is assigned to the return variable.

After every instruction is analysed, the final return is modified to ensure that it returns the return variable.

As an example, consider the early-exit part of the vulnerability in listing 3.2, where the parameter "pw" is a secret. Here, it is necessary to remove two return statements. However, the result of the execution of the method must be the same. For that, the value returned will be saved in a variable and that variable is returned at the end of the method execution. The result of the correction of the early-exit part of the vulnerability of the method can be seen in listing 3.3. Here, it is created the variable *$1*, which is the return variable and it is initialised with the value returned in the last return of the method. Then, where there was a return statement, it is replaced with an assignment of the value being returned to *$1*. Then, when the tool reaches the final return, it simply changes the value returned to be the variable *$1*.

Once this was done, the tool was tested, using EvoSuite, against every example of an early-exit timing side-channel in DifFuzz examples. In that test, the tool was able to correct several of the examples, in a way that the produced code was according to what was expected.

While executing the tests it became clear that the correction created problems of *IndexOutOfBounds* and other exceptions. This happens because some of the returns removed were protecting future instructions, e.g. checking if the index used to access an array is valid. So, the correction process needed some modifications.

These modifications aim at ensuring that even after removing a return statement, certain statements remain protected. So every time the tool replaces a return statement from inside an ‘if’ statement, its condition is analysed. If that condition checks if a variable is null, that variable is stored in a list. If that condition, contains a variable read then that variable is stored alongside the original condition. Whenever any of those variables is accessed again in a future statement a new condition is added. If that variable is accessed in the stopping condition of a cycle then the whole body of the cycle becomes the ‘then’ block of a new ‘if’ statement where the condition will be all the conditions found where the variable used in the stopping condition was part of. That ‘if’ statement then becomes the body of the cycle. If the variable read is part of the value assigned in the definition of a new variable, then that local variable statement will become the statement of the ‘then’ block of a new ‘if’ statement where the condition is all the previous conditions found that used the variable. However, since the condition to be added was used as an exit condition, to protect the variable, that condition must be negated. If an ‘if’ statements´ condition uses an array read, then before that array read, the index used to access the array is validated by checking if it is less then the array size.

As an example, consider the early-exit part of the vulnerability in listing 4.5, where the parameter “src” is a secret. Here the first ‘if’ statement checks if the value of *srcLength* is greater than the value of *totalLength*. If so, returns immediately. However, to correct the early-exit part of the timing side-channel vulnerability, that return alongside all returns need to be removed. This would break the functionality of the code since the value of *padLength* should not be negative. To prevent this, the tool saves the condition of the first ‘if’ statement and when in the creation of the variable *padLength* the method tries to use variables that were protected, the tool creates a new ‘if’ statement where the condition is the combination of the negation of the conditions where that variables were part of. In this case, since the condition is the same for both variables, the new condition is simply the negation of that condition. This way, the tool tries to ensure that no exceptions and that no breaks of the method logic exist. The corrected version of the method can be seen in listing 4.6.

```
public static final String pad_unsafe(String src, char padChar, boolean rightPad,
     int totalLength) {
    int srcLength = src.length();
    if (srcLength >= totalLength) {
        return src;
    }
    int padLength = totalLength - srcLength;
    StringBuilder sb = new StringBuilder(padLength);
    for (int i = 0; i < padLength; ++i) {
        sb.append(padChar);
    }
    if (rightPad) {
        return src + sb.toString();
    } else {
        return sb.toString() + src;
    }
}
```

Listing 4.5: Vulnerable Method

```
public static final String pad_unsafe$Modification(String src, char padChar,
     boolean rightPad, int totalLength) {
    int $5 = 0;
    int $4 = src.length();
    int $3 = totalLength;
    String $2;
    String $1;
    int srcLength = src.length();
    StringBuilder $6 = new StringBuilder(srcLength);
    if (srcLength >= totalLength) {
        $1 = src;
    } else {
        $2 = src;
    }
    int padLength = 0;
    if (!(srcLength >= totalLength)) {
        padLength = totalLength - srcLength;
    } else {
        $5 = $3 - $4;
    }
    StringBuilder sb = new StringBuilder(padLength);
    for (int i = 0; i < totalLength; ++i) {
        if (i < padLength) {
            sb.append(padChar);
        } else {
            $6.append(padChar);
        }
    }
    if (rightPad) {
        $1 = src + sb.toString();
    } else {
        $1 = sb.toString() + src;
    }
    return $1;
}
```

Listing 4.6: Vulnerable Method after early-exit correction

In the end, a new version of the class that contains the vulnerable method is created. This version is similar to the original version, except that it contains an extra method called *VulnerableMethod-Name$Modification*. So, if the users want to use the corrected version, they must replace the original method for the corrected method.

The tool corrects the method indicated in the Driver used for DifFuzz and does not check if the vulnerability is instead in a method invoked by the indicated method. As such, the user should ensure that the Driver points to the vulnerable method.

4.4 Correction of Control-Flow Based Timing Side-Channel Vulnerabilities

Control-Flow based timing side-channel vulnerabilities happen when based on the value of a secret the executing time of a method differs, either because it executes more iterations of a cycle or because it takes a path on a branch that takes longer to execute. So, to repair the control-flow based timing side-channel vulnerability it is necessary to ensure that no cycle iterations depend on a secret and that in a conditional statement each branch execution time is similar if the condition depends on a secret.

To correct this type of timing side-channel vulnerability the method secret arguments are kept in a list, and the public arguments are kept on another. Then the tool analysis each instruction of the method. When it finds a cycle, it checks its stopping condition. If it uses a secret, the tool tries to find from the list of public arguments an argument of the same type as the secret and replaces it. This way the number of instructions the cycle performs is independent of a secret.

When the tool finds an ‘if’ statement it checks its condition. If the condition uses a secret, then the tool creates a modified version of the ‘then’ and ‘else’ blocks. Then, the modified version of the ‘then’ block is added to the ‘else’ block and the modified version of the ‘else’ block is added to the ‘then’ block. The modified version contains all instructions that do not depend on any variable used in the condition. In those instructions, if there is an assignment, it is created a new variable so that it can be assigned instead of the original one. It is kept a record of all new variables and the variables they are replacing.

Then, if another instruction of the current branch uses a replaced variable, that variable in that new instruction is replaced for the previously created variable.

After analysing every instruction of the vulnerable method, the tool creates a copy of the method with the modifications performed. And, for every variable created as a replacement for an existing variable, their creation instruction is added to the method, except for when the new variable was created to replace a variable being created. Those variables creation instructions are added to as early as possible in the method, the only restriction is that when the variable created is created with an initial value, every variable that is part of the creation must be created before that variable is created. Meaning, that the new variable is placed in a position of the method where all the variables they depend on exist.

As an example, take the code in listing 4.7. Here, the method has a control-flow based timing side-channel vulnerability, since its execution time depends on the value of a secret, in this case the parameter “taint”. The ‘then’ and ‘else’ block, of the ‘if’ statement has different instructions with different execution time. As such, to correct this vulnerability the tool will modify the instructions of both blocks and add the modified version of the ‘then’ block to the ‘else’ block and the modified version of the ‘else’ block to the ‘then’ block. The difference in the instructions is that the assignment is not made to the same variables, so as not to ruin the value of the original variable. As a result, the tool produces the code in listing 4.8.

```java
public static boolean array_unsafe(int a[], int taint) {
    System.out.println(a.length);
    int t;
    if (taint < 0) {
        int i = a.length-1;
        while (i >= 0) {
            t = a[i];
            i--;
        }
    } else {
        int i = 0;
        t = a[i] / 2;
        i = a.length;
    }
    return false;
}
```

Listing 4.7: Example of method with control-flow based timing side-channel vulnerability

```java
public static boolean array_unsafe$Modification(int[] a, int taint) {
    int $2;
    System.out.println(a.length);
    int t;
    if (taint < 0) {
        int $3 = 0;
        $2 = a[$3] / 2;
        $3 = a.length;
        int i = a.length - 1;
        while (i >= 0) {
            t = a[i];
            i--;
        }
    } else {
        int $1 = a.length - 1;
        while ($1 >= 0) {
            $2 = a[$1];
            $1--;
        }
        int i = 0;
        t = a[i] / 2;
        i = a.length;
    }
    return false;
}
```

Listing 4.8: Corrected version of method with control-flow based timing side-channel vulnerability

Then, the tool was executed against all examples of DifFuzz that contain a control-flow based timing side-channel vulnerability. The tool was capable of correcting some examples. Using EvoSuite, tests were created for those examples that showed that the correction is semantically correct. DifFuzz was executed on the corrected version of those examples showing that the tool corrected some examples.

However, this first round of tests also showed that the approach to the correction has some limitations.

One of the limitations is that a variable used in a stopping condition or condition of an ‘if’ statement might not be a secret argument but a variable whose value is influenced by the value of a secret. To also correct those examples, every time a secret is used to provide the value of a variable, that variable also becomes a secret and is added to the list.

Another limitation happens when the ‘then’ block of an ‘if’ statement is an ‘if’ statement with an invocation as a condition and a variable used in the condition of the former is also used in the condition of the latter. When that happens, it is necessary to modify the condition of the inner ‘if’ statement. For that, every invocation on the condition of the inner ‘if’ statement, is turned into a series of assignments except for the invocation that returns a boolean since that one is used as the condition of the ‘if’ statement to add to the modified ‘else’ block of the outer ‘if’ statement. As an example take the code in listing 4.9, where the secret is the variable “map”. Here the condition of the first ‘if’ statement is an invocation that uses the variables *map* and *u*. Then, when the condition is met, another ‘if’ statement is executed and in its condition, there is a series of invocations that uses both variables used in the previous ‘if’ statement. The tool to try and correct the vulnerability needs to modify the ‘then’ block and add it to the ‘else’ block. To do this in this case, it needs to remove the series of invocations from the condition of the ‘if’ statement and transform them into a series of invocation statements. The only invocation that can continue in the condition is the one where the result is a boolean. One of the invocations can not be added to the ‘else’ block, because it uses variables that were used in the condition of the outer ‘if’ statement. Analysing this invocation, the tool knows that it returns a String, and another of the invocations in the condition also returns a String. As such, the tool duplicates that invocation to replace the invocation that can not happen in the ‘else’ block. As a result, the tool produces the code in listing 4.10.

```java
public static boolean login_unsafe(String u, String p) {
    boolean outcome = false;

    if (map.containsKey(u)) {
        if (map.get(u).equals(md5(p))) {
            outcome = true;
        }
    }

    return outcome;

}
```

Listing 4.9: Example of method with invocations in condition to modify

```java
public static boolean login_unsafe$Modification(String u, String p) {
    boolean $4;
    boolean $1 = false;
    boolean outcome = false;
    if (map.containsKey(u)) {
        if (map.get(u).equals(md5(p))) {
            outcome = true;
        } else {
            $1 = true;
        }
    } else {
        String $2 = md5(p);
        String $3 = md5(p);
        if ($2.equals($3)) {
            $1 = true;
        } else {
            $4 = true;
        }
    }
    return outcome;
}
```

Listing 4.10: Corrected version of method with invocations in condition

The other limitation happens when replacing a variable, if that variable is an array, then the tool would only replace the array target and not its index. As such, if the variable used as the index of the array was replaced, then when modifying that array access, it is necessary to replace the variable used as the index. Another limitation with arrays is when a variable used in the condition of an 'if' statement is an array, then the array to be added to the other branch, should be a newly created array. That array will be created with the size of 1 and every access done to the array will be done to its first index.

After all these modifications the tool was able to successfully correct a few more examples of a control-flow based timing side-channel vulnerability. However, the tool still is not able to correct all examples.

4.5 Correction of Mixed Timing Side-Channel Vulnerabilities

Sometimes a method has an early-exit and a control-flow based timing side-channel vulnerability. For that reason, the tool tries to correct both types of vulnerability in a single execution. If the method has more than one return statement the tool tries to repair an early-exit timing side-channel vulnerability producing a modified version of the method. Then, the tool tries to correct the control-flow based timing side-channel vulnerability in the modified version of the method, producing the final version of the method. This means that each module responsible for correcting a type of timing side-channel vulnerability must return its modified version of the method. Since both repair processes create new variables in the method, and a method can't have two variables with the same name, the naming of a variable is global to the tool and it keeps a record of the name of the last variable.

Once the tool started correcting both types of timing side-channel vulnerability it corrected more examples of DifFuzz.

5 Evaluation

Contents

5.1 Dataset Used 65

5.2 Semantic Correction 65

5.3 Vulnerability Correction 66

5.4 Dataset Examples 69

In this chapter, we describe how the developed tool was evaluated. The evaluation consists in ensuring that the refactored code is semantically correct and that it has no side-channel vulnerabilities. This chapter presents both types of evaluation, explaining how they are done as well as why they are necessary.

5.1 Dataset Used

Since this project is inspired by the work developed for the tool DifFuzz it was decided that DifFuzz would be used to help evaluate the tool. For that reason, the examples of DifFuzz were chosen to test the tool. The dataset of DifFuzz contains 32 examples. However, one of the examples suffers from a size side-channel, and in some, it was not possible to understand how they are vulnerable. Moreover, in other examples the vulnerability did not follow the template of vulnerable method considered in this project (more information can be seen in section 5.4.1). As such, to test the tool only 25 of those examples were used. Those examples were categorized according to the type of vulnerability. Two of the examples suffer from early-exit timing side-channel vulnerability. Eight of the examples contain a control-flow based timing side-channel vulnerability. While the rest, meaning 15 examples, have a mixed timing side-channel vulnerability.

5.2 Semantic Correction

To correct vulnerabilities in a method it is necessary to modify it. This modification can 'break' the method, in the sense that its output will no longer be the same for the same input. As such, it is important that after any modifications to a method, the same is tested to ensure that its functionality remains. The same is true for modifications made by the tool. Although the tool does not test the modifications automatically, the user of the tool should ensure that the method created by the tool still works like the original method.

During the development of the tool, the application examples used by the authors of DifFuzz were used to ensure that the tool was capable of correcting a vulnerability. However, these examples did not include tests, so it was not possible to ensure the correction kept the functionality of the method. One solution was to create unit tests for every example of DifFuzz. The problem with this solution is that the process of creating tests is long and prone to errors. This is even worse, given that the examples are small sections of huge applications, which would make the development of tests a difficult task. So, a different solution was found, to create tests automatically. This would ensure that the creation of tests is faster and less prone to error. To create tests automatically it was chosen the tool EvoSuite [25]. This tool generates unit tests automatically for Java software and has a plugin allowing its integration with Maven.

So to test if the solution created by the tool is semantically correct, it is created a maven project for the examples of DifFuzz to test. To these projects, EvoSuite is added as a dependency alongside the original dependencies of the application in the .pom file. Then the project is compiled and the necessary commands of EvoSuite, to create and add tests to the projects, are executed. These tests are created and first run on the original code, the vulnerable one, to see if the tests are correct. Then, the vulnerable method is replaced for the method created by the tool and the tests are executed again. If all tests pass, then the solution created by the tool to correct the timing side-channel vulnerability is semantically correct.

When creating tests for some examples, EvoSuite created tests that failed in the original code. Since EvoSuite created tests for that specific code, this was taken as a bug and those tests ignored. Despite this, those failing tests were still executed on the repaired version of the method and some of them passed.

More details about the results of the semantic correction of the examples created by the tool can be seen in the next section.

5.3 Vulnerability Correction

Once the tool repaired a vulnerable method and that repair is shown to be semantically correct it is necessary to verify if the repair produced by the tool repaired the vulnerability. Since this project was inspired by DifFuzz, that is the tool used to verify if there is any timing side-channel vulnerability. This execution follows the scripts created by the authors of DifFuzz, the difference being that now besides the safe and unsafe versions provided by the authors it is also tested the corrected version. The corrected version is a copy of the unsafe version where the vulnerable method is replaced by the corrected version created by the tool.

This test was performed in a remote server with a 32-processor Intel Xeon Silver 4110 at 2.10GHz with 64GB of RAM running Debian Linux 10 and each version of the example ran for 2,5 hours. The results of the execution of tests created by EvoSuite and of the analysis performed by DifFuzz can be seen in Table 5.1.

That table shows that out of 32 examples, the tool was capable of correcting 14 examples which gives a success rate of 43,7%. That is not a great number, but in those 32 examples, some are ignored, either because the type of vulnerability is unknown or because its vulnerability is not able to be corrected by the tool like the example *Themis TourPlanner* whose vulnerability is related to the size of the response. So if it is only considered the examples that were attempted to correct then out 25 examples, the tool successfully corrected 14 of them, making for a success rate of 56%. That is still not a great number but it is more reassuring. Out of all the examples the tool tried to correct not all of them are semantically correct, meaning that the code lost some of the functionality after the repair. If it is only considered semantically correct examples, then the total of examples is 22, which makes a success rate of 63,6%.

Dataset name	**Has secure version?**	**Type**	**Correction Attempted**	**Semantically Correct**	**Correct Vulnerability**
Apache FtpServer Clear	Yes	Mixed	Yes	No	-
Apache FtpServer Md5	Yes	Early-Exit (If dependant)	Yes	No	-
Apache FtpServer Salted Encrypt	No	Unknown	No	-	-
Apache FtpServer Salted	Yes	Mixed	Yes	No	-
Apache FtpServer StringUtils	Yes	Mixed	Yes	Yes	Yes
Blazer Array	Yes	Control-Flow	Yes	Yes	Yes
Blazer Gpt14	Yes	Control-Flow	Yes	Yes	No
Blazer K96	Yes	Control-Flow	Yes	Yes	Yes
Blazer LoopAndBranch	Yes	Control-Flow (ignored)	No	-	-
Blazer Modpow1	Yes	Control-Flow	Yes	Yes	Yes
Blazer Modpow2	Yes	Unknown	No	-	-
Blazer PasswordEq	Yes	Early-Exit (If dependant)	Yes	Yes	Yes
Blazer Sanity	Yes	Mixed	Yes	Yes	Yes
Blazer StraightLine	Yes	Control-Flow	Yes	Yes	Yes
Blazer UnixLogin	Yes	Control-Flow	Yes	Yes	Yes
Example PWCheck	Yes	Mixed	Yes	Yes	Yes
GitHub AuthmReloaded	Yes	Mixed	Yes	Yes	Yes
STAC Crime	No	Unknown	No	-	-
STAC Ibasys	No	Control-Flow	Yes	Yes	No
Themis Boot-Stateless-Auth	Yes	Mixed	Yes	Yes	No
Themis Dynatable	No	Mixed	Yes	Yes	No
Themis GWT Advanced Table	No	Unknown	No	-	-
Themis Jdk	Yes	Mixed	Yes	Yes	Yes
Themis Jetty	Yes	Mixed	Yes	Yes	Yes
Themis OACC	No	Mixed	Yes	Yes	Yes
Themis OpenMrs-Core	No	Unknown	No	-	-
Themis OrientDb	Yes	Mixed	Yes	Yes	No
Themis Pac4j	Yes	Control-Flow	Yes	Yes	Yes
Themis PicketBox	Yes	Mixed	Yes	Yes	No
Themis Spring-Security	Yes	Mixed	Yes	Yes	No
Themis Tomcat	Yes	Mixed	Yes	Yes	No
Themis TourPlanner	Yes	Size Side-Channel	No	-	-

Table 5.1: Results of tool

Some further analysis can be done of the obtained results. For example, if it is analysed only the examples with an early-exit timing side-channel vulnerability, the result is the table 5.2. Here, it is clear that out of 2 examples the tool was only capable of correcting successfully one of them. Even more, in one of the examples, the correction created by the tool is not semantically correct.

Dataset name	**Has secure version?**	**Type**	**Correction Attempted**	**Semantically Correct?**	**Correct Vulnerability?**
Apache FtpServer Md5	Yes	Early-Exit (If dependant)	Yes	No	-
Blazer PasswordEq	Yes	Early-Exit (If dependant)	Yes	Yes	Yes

Table 5.2: Early-Exit results

The same analysis can be done for examples with control-flow based timing side-channel vulnerabilities, which result can be seen in table 5.3. Here, the tool was capable of correcting 6 out of 9 examples, giving a success rate of 66,7%. Out of the 9 examples, one of them is ignored because the vulnerability is not understood. So, if in the results that example is also ignored, then the tool corrected 6 out of 8 examples, making a success rate of 75%. As opposed to the correction of early-exit vulnerabilities, none of the corrections proposed by the tool for this type of problem breaks the application, meaning that all repairs are semantically correct.

Dataset name	Has secure version?	Type	Correction Attempted	Semantically Correct?	Correct Vulnerability?
Blazer Array	Yes	Control-Flow	Yes	Yes	Yes
Blazer Gpt14	Yes	Control-Flow	Yes	Yes	No
Blazer K96	Yes	Control-Flow	Yes	Yes	Yes
Blazer LoopAndBranch	Yes	Control-Flow (ignored)	No	-	-
Blazer Modpow1	Yes	Control-Flow	Yes	Yes	Yes
Blazer StraightLine	Yes	Control-Flow	Yes	Yes	Yes
Blazer UnixLogin	Yes	Control-Flow	Yes	Yes	Yes
STAC Ibasys	No	Control-Flow	Yes	Yes	No
Themis Pac4j	Yes	Control-Flow	Yes	Yes	Yes

Table 5.3: Control-Flow results

The final type of timing side-channel vulnerability is Mixed, and its results can be seen in table 5.4. In this case, the tool successfully corrected 7 out of 15 examples, making for a success rate of 46,7%. In this case, the correction proposed by the tool breaks the application in two examples, which means that the result of the execution of the tool in two examples is not semantically correct. So, the tool has a 13,3% chance of producing a solution that breaks the application and a 46,7% chance of correcting the vulnerability.

Dataset name	Has secure version?	Type	Correction Attempted	Semantically Correct?	Correct Vulnerability?
Apache FtpServer Clear	Yes	Mixed	Yes	No	-
Apache FtpServer Salted	Yes	Mixed	Yes	No	-
Apache FtpServer StringUtils	Yes	Mixed	Yes	Yes	Yes
Blazer Sanity	Yes	Mixed	Yes	Yes	Yes
Example PWCheck	Yes	Mixed	Yes	Yes	Yes
GitHub AuthmReloaded	Yes	Mixed	Yes	Yes	Yes
Themis Boot-Stateless-Auth	Yes	Mixed	Yes	Yes	No
Themis Dynatable	No	Mixed	Yes	Yes	No
Themis Jdk	Yes	Mixed	Yes	Yes	Yes
Themis Jetty	Yes	Mixed	Yes	Yes	Yes
Themis OACC	No	Mixed	Yes	Yes	Yes
Themis OrientDb	Yes	Mixed	Yes	Yes	No
Themis PicketBox	Yes	Mixed	Yes	Yes	No
Themis Spring-Security	Yes	Mixed	Yes	Yes	No
Themis Tomcat	Yes	Mixed	Yes	Yes	No

Table 5.4: Mixed results

A final table with all examples considered invalid can be seen in table 5.5.

Analysing the results obtained from the tables it is clear that the tool is less effective when correcting early-exit timing side-channel vulnerabilities, both in the probability of correcting the vulnerability and of creating a correction that breaks the application. However, the tool is most successful when cor-

Dataset name	**Has secure version?**	**Type**	**Correction Attempted**	**Semantically Correct?**	**Correct Vulnerability?**
Apache FtpServer Salted Encrypt	No	Unknown	No	-	-
Blazer LoopAndBranch	Yes	Control-Flow (ignored)	No	-	-
Blazer Modpow2	Yes	Unknown	No	-	-
STAC Crime	No	Unknown	No	-	-
Themis GWT Advanced Table	No	Unknown	No	-	-
Themis OpenMrs-Core	No	Unknown	No	-	-
Themis TourPlanner	Yes	Size Side-Channel	No	-	-

Table 5.5: Invalid examples

recting control-flow based timing side-channel vulnerabilities, correcting the majority of the examples. When correcting mixed timing side-channel vulnerabilities the tool has a good chance of repairing the vulnerability but can also create a solution that breaks the application.

These results show the importance of having unit tests for the application before using the tool and that while the tool still needs to be improved it can be used as a guideline to show developers how to correct a vulnerability.

5.4 Dataset Examples

In this subsection, the invalid examples for which corrections were not attempted are described.

5.4.1 Invalid Examples

The first example is the one named **apache_ftpserver_salted_encrypt**. Unlike the majority of DifFuzz examples, this one does not contain a safe example. Meaning that the authors of DifFuzz did not provide a correct version of the code. Although that does not make an example invalid, in this case, makes it harder to understand the vulnerability. In this example, the method referred to as the vulnerable one is called *encrypt*, where the secret is the second argument and can be seen in listing 5.1. Analysing the method, it becomes clear that it does not contain an early-exit timing side-channel vulnerability, since there is no condition dependant on the secret with an exit-condition; in fact, there is no condition at all. And it does not suffer from a control-flow based timing side-channel vulnerability since in the only cycle present in the method the number of interactions is fixed number.

```
public /*private*/ String encrypt(String password, String salt) {
    long tmpCost = Mem.instrCost;
    String hash = salt + password;
    for (int i = 0; i < HASH_ITERATIONS; i++) {
        hash = EncryptUtils.encryptMD5(hash);
    }
    return salt + ":" + hash;
}
```

Listing 5.1: Vulnerable method encrypt

Another example is **blazer_loopandranch**. Unlike the previous one, for this one, it is provided a safe version of the vulnerable method and in this case, the vulnerability is also understood as a control-flow based timing side-channel vulnerability. The problem in this example, however, is with the correction. Comparing the safe and unsafe versions, available in listing 5.2, of the method it is clear that the only difference is in the first instruction of the 'else' block. In the safe version the instruction 'taint = taint - 10' is changed to 'taint = taint + 10'. This modification requires a more detailed analysis than the one that the tool is able. To correct a control-flow based timing side-channel vulnerability, the tool simply tries to ensure that no cycle stopping condition depends on a secret and that for any branch execution where the condition depends on a secret, the execution time is similar regardless of the value of the secret. Neither of those attempts will correct this example as intended.

```
public static boolean loopAndbranch_unsafe(int a, int taint) {
    int i = a;

    if (taint < 0) {
        while (i > 0) {
            i--;
        }
    } else {
        taint = taint - 10;

        if (taint >= 10) {
            int j = a;
            while (j > 0) {
                j--;
            }
        } else {
            if (a < 0) {
                int k = a;
                while (k > 0)
                    k--;
            }
        }
    }

    return true;
}
```

Listing 5.2: Vulnerable method loopAndBranch

The next example is **blazer_modpow2**. Like the previous one, for this example a safe version of the vulnerable method is available. Analysing the vulnerable method, available in listing 5.3, it becomes clear that it does not contain an early-exit timing side-channel vulnerability. However, since the secrets of the method are the arguments *exponent* and *width*, the method may contain a control-flow based timing side-channel vulnerability. Analysing the 'if-else' block it becomes clear that the same instructions are executed regardless of the value of the secret. So the only way for a vulnerability would be the fact that the stopping condition of the cycle depends on a secret. However, in the safe version, the stopping condition of the cycle is the same, the only difference is two instructions in the 'if-else' block. That means that the vulnerability present in the method is not something the tool could repair and for that reason, this example is ignored.

```
//top-level modPow method: inline the custom implementation of fastMultiply
public static BigInteger modPow2_unsafe(BigInteger base, BigInteger exponent,
     BigInteger modulus, int width) {
    BigInteger r0 = BigInteger.valueOf(1);
    BigInteger r1 = base;
    //        int width = exponent.bitLength(); // use width parameter because
        bitlength is wrong for 0
    for (int i = 0; i < width; i++) {
        if (!exponent.testBit(width - i - 1)) {
            r1 = fastMultiply_inline(r0, r1).mod(modulus);
            r0 = r0.multiply(r0).mod(modulus);
        } else {
            r0 = fastMultiply_inline(r0, r1).mod(modulus);
            r1 = r1.multiply(r1).mod(modulus);
        }
    }
    return r0;
}
```

Listing 5.3: Vulnerable method modPow2

Another example is **stac_crime**. For this example, no safe version is provided which makes it harder to understand the vulnerability. Analysing the code, available in listing 5.4, becomes clear that there is no early-exit timing side-channel vulnerability since the only exit point of the method is the final instruction. However, this example might suffer from control-flow based timing side-channel since the stopping condition of the cycle is dependant on the value of the secret. However, in this method, the secret is also its only argument and the only value the method can depend on. Although not clear, that is the best explanation of what type of timing side-channel vulnerability this method has. However, even if the vulnerability is a control-flow based timing side-channel, the tool would not be able to correct it, since in this case, it would try to replace the secret used in the stopping condition of the cycle for a public argument of the method, which in this case does not exist.

```
public static byte[] compress(final byte[] in) throws IOException {
    StringBuffer mSearchBuffer = new StringBuffer(1024);
    final ByteArrayInputStream stream = new ByteArrayInputStream(in);
    final InputStreamReader reader = new InputStreamReader((InputStream)stream);
    final Reader mIn = (Reader)new BufferedReader((Reader)reader);
    final ByteArrayOutputStream oStream = new ByteArrayOutputStream();
    final OutputStreamWriter writer = new OutputStreamWriter((OutputStream)
        oStream);
    final PrintWriter mOut = new PrintWriter((Writer)new BufferedWriter((Writer)
        writer));
    String currentMatch = "";
    int matchIndex = 0;
    int tempIndex = 0;
    int nextChar;
    while ((nextChar = mIn.read()) != -1) {
        tempIndex = mSearchBuffer.indexOf(currentMatch + (char)nextChar);
        if (tempIndex != -1) {
            currentMatch += (char)nextChar;
            matchIndex = tempIndex;
        }
        else {
            final String codedString = new StringBuilder().append("~").append(
                matchIndex).append("~").append(currentMatch.length()).append("~")
                .append((char)nextChar).toString();
            final String concat = currentMatch + (char)nextChar;
            if (codedString.length() <= concat.length()) {
                mOut.print(codedString);
                mSearchBuffer.append(concat);
                currentMatch = "";
                matchIndex = 0;
            }
            else {
                for (currentMatch = concat, matchIndex = -1; currentMatch.length
                    () > 1 && matchIndex == -1; currentMatch = currentMatch.
                    substring(1, currentMatch.length()), matchIndex =
                    mSearchBuffer.indexOf(currentMatch)) {
                    mOut.print(currentMatch.charAt(0));
                    mSearchBuffer.append(currentMatch.charAt(0));
                }
            }
            if (mSearchBuffer.length() <= 1024) {
                continue;
            }
            mSearchBuffer = mSearchBuffer.delete(0, mSearchBuffer.length() -
                1024);
        }
    }
    if (matchIndex != -1) {
        final String codedString = new StringBuilder().append("~").append(
            matchIndex).append("~").append(currentMatch.length()).toString();
        if (codedString.length() <= currentMatch.length()) {
            mOut.print(new StringBuilder().append("~").append(matchIndex).append(
                "~").append(currentMatch.length()).toString());
        }
        else {
            mOut.print(currentMatch);
        }
    }
    mIn.close();
    mOut.flush();
    final byte[] bytes = oStream.toByteArray();
```

```java
    mOut.close();
    return bytes;
}
```

Listing 5.4: Vulnerable method compress

The next example is **themis_gwt_advanced_table**. This also does not contain a safe version of the vulnerable method. And it was also not possible to understand what type of vulnerability there is and how to correct it. Analysing the code, available in listing 5.5, it is possible to conclude that the method does not have an early-exit timing side-channel vulnerability since it only contains an exit point. Given that the first two arguments, *startRow* and *rowsCount*, are the secrets and that *rowsCount* is used as part of the stopping condition of the outer cycle, this method may suffer from control-flow based timing side-channel vulnerability. However, even if that is the case, the stopping condition of the outer cycle depends on the size of the array and as such can not be changed without breaking the application. For that reason, this method is ignored.

```java
public String[][] getRows_unsafe(int startRow, int rowsCount, DataFilter[]
    filters, String sortColumn,
            boolean sortingOrder) {
    User[] rowsData = getRowsData(startRow, rowsCount, filters, sortColumn,
        sortingOrder);
    int columnsCount = this.columns.length;
    String[][] rows = new String[rowsCount][columnsCount];
    for (int row = 0; row < rowsCount; row++) {
        for (int col = 0; col < columnsCount; col++) {
            String columnName = this.columns[col].getName();
            rows[row][col] = ReflectionUtils.getPropertyStringValue(rowsData[row
                ], columnName);
        }
    }
    return rows;
}
```

Listing 5.5: Vulnerable method getRows

Another example is **themis_openmrs-core**. Like the previous example, this one also does not contain a safe version of the vulnerable method. The method referenced in the Driver is available in listing 5.6, where the secret is the argument *hashedPassword* and is used to determine if an exception should be thrown and is compared to the result of the invocation of other methods with a public argument. With that, it is safe to assume that the vulnerability is not in any of the other methods since they do not get access to the secret. And the condition to throw the exception is necessary since without it the method can not successfully execute. Therefore, this method is ignored.

```java
public static boolean hashMatches(String hashedPassword, String passwordToHash) {
    if (hashedPassword == null || passwordToHash == null) {
        throw new APIException("password.cannot.be.null", (Object[]) null);
    }

    return hashedPassword.equals(encodeString(passwordToHash))
            || hashedPassword.equals(encodeStringSHA1(passwordToHash))
            || hashedPassword.equals(incorrectlyEncodeString(passwordToHash));
}
```

Listing 5.6: Vulnerable method hashMatches

The final example is **themis_tourplanner**. Unlike the previous example, this one contains a safe version of the method. Analysing the method, available in listing 5.7, where both arguments are secrets it becomes clear that not only does this method does not suffer from early-exit or control-flow based timing side-channel vulnerability but is does not contain a timing side-channel vulnerability but a size side-channel vulnerability. That means this example can not be used with the tool.

```java
public void doGet(HttpServletRequest req, final HttpServletResponse res) throws
     ServletException, IOException {
    List<GHPoint> points = this.getPoints(req, "point");
    TourResponse tourRsp = this.tourCalculator.calcTour(points);
    List<String> list = this.tourSerializer.toList(tourRsp);
    PrintWriter writer = res.getWriter();
    String listStr = list.toString();
    writer.append(listStr);

    // Safe version
    if (safe) {
        int len = listStr.length();
        assert (len <= MSG_FULL_LENGTH);
        for (int i = len; i < MSG_FULL_LENGTH; ++i)
```

```
            writer.append(" ");
    }
}
```

Listing 5.7: Vulnerable method doGet

5.4.2 Unsuccessful Examples

When using the tool on the examples deemed valid, the tool was still not able to successfully correct all examples. In some, the correction although did not completely repair the vulnerability, it showed how to correct that vulnerability and sometimes, even reduced the effects of the vulnerability, by reducing the time difference of the method execution. As such, in those examples, the user could look at the code and try to improve it to repair the vulnerability. The major problem is with some examples where the tool created a faulty version of the method that did not compile. That means, that the user could not use the tools' method as a replacement for the original method, only as a source of inspiration to try to manually repair the vulnerability.
As a warning for the method templates that the tool was not able to repair is now explained each failing example and why it failed.

5.4.2.A Faulty Examples

The first example of a method that the tool could not correct is **apache_ftpserver_clear**. The solution created by the tool can be seen in listing 5.8. The problem with this solution is in the 'else' block inside the 'if' block, the stopping condition of the cycle involves a variable named *$3*, and there that variable was not created yet. The variable *$3* is supposed to be a replacement for the variable *n*, and since the variable *n* is part of the condition of the 'if' block it can not be used in the corresponding 'else' block and so needs to be replaced. The problem is that the variable *n* is replaced by *$3*, even before the tool starts to modify that 'if' block. And when the variable is replaced, *$3* is expected to be created in the outer 'else' block, and so its creation will not be added at a point where the cycle can access it, making the application not able to compile. One possible solution would be to, whenever a variable is replaced, the creation of the new variable would be added as the first instruction after the creation of the replaced variable. The problem with this solution is that this would add several extra instructions to the code and would create several instructions with the same name. All that would make it so that it would not be possible to minimize the execution time of the branches.

```
public boolean isEqual_unsafe$Modification(String thisObject, Object otherObject)
     {
    boolean $2 = false;
    boolean $1 = false;
    if (thisObject == otherObject) {
        $1 = true;
    } else {
        $2 = true;
    }
    if (otherObject instanceof String) {
        String anotherString = ((String) (otherObject));
        int n = thisObject.length();
        if (n == anotherString.length()) {
            char[] v1 = thisObject.toCharArray();
            char[] v2 = anotherString.toCharArray();
            int i = 0;
            $1 = true;
            while ((n--) != 0) {
                if (((i < v1.length) && (i < v2.length)) && (v1[i] != v2[i])) {
                    $1 = false;
                } else {
                    $2 = false;
                }
                i++;
            }
        } else {
            char[] $4 = thisObject.toCharArray();
            int $5 = 0;
            $2 = true;
            while (($3--) != 0) {
                $5++;
            }
        }
    } else {
        int $3 = thisObject.length();
        char[] $4 = thisObject.toCharArray();
        int $5 = 0;
        $2 = true;
        while (($3--) != 0) {
            $5++;
        }
    }
    return $1;
}
```

Listing 5.8: Vulnerable method isEqual

Next are the examples **apache_ftpserver_md5** and **apache_ftpserver_salted**. In these two examples, the vulnerable method is the same and so the correction of the tool and the reason it fails is the same. As such, only the solution created by the tool to one of the examples is available in listing 5.9. The problem with this solution is that it introduces 'ArrayIndexOutOfBoundsException'. This is a problem with trying to correct early-exit timing side-channel vulnerabilities because some times an early-exit of a method is there to protect the access to an array. So, when correcting the vulnerability by removing that return, the tool needs to ensure that any access to an array is protected. To do this, every time a return instruction is replaced for an assignment, the tool records a combination between any variable used in the condition and the condition itself. That way any time that variable is used in an instruction, that instruction is protected by adding a combination of the negation of the conditions the variable was used on. The problem in these examples is that the variables used in the condition of an 'if' statement are not the same as the ones used to access the array in the first two instructions of the cycle. As such, the tool will not add an 'if' statement as protection. A possible way to resolve this issue would be to know that the variables used as the index to access the arrays *ta* and *pa* are the same as some of the variables used in the condition of the 'if' statement outside the cycle. The problem with this is that the tool uses the tool *spoon* to access the instructions of the method and that tool do not have the information or a way to indicate that the variables are the same.

```
public boolean regionMatches$Modification(String thisValue, boolean ignoreCase,
     int toffset, String other, int ooffset, int len) {
    boolean $2 = true;
    boolean $1 = true;
    char[] ta = thisValue.toCharArray();
    int to = toffset;
    char[] pa = other.toCharArray();
    int po = ooffset;
    if ((((ooffset < 0) || (toffset < 0)) || (toffset > (((long) (thisValue.
        length())) - len))) || (ooffset > (((long) (other.length())) - len))) {
            $1 = false;
        } else {
            $2 = false;
        }
        while ((len--) > 0) {
            char c1 = ta[to++];
            char c2 = pa[po++];
            if (c1 == c2) {
                continue;
            }
            if (ignoreCase) {
                char u1 = Character.toUpperCase(c1);
                char u2 = Character.toUpperCase(c2);
                if (u1 == u2) {
                    continue;
                }
                if (Character.toLowerCase(u1) == Character.toLowerCase(u2)) {
                    continue;
                }
            }
            $1 = false;
        }
        return $1;
    }
```

Listing 5.9: Vulnerable method regionMatches

5.4.2.B Unsuccessful Correction

For some examples, the tool was capable of creating a version of the vulnerable method that compiled and passed all unit tests available. However, the solution was not free of timing side-channel vulnerabilities. As such, those examples are now referenced and the problem with the solution is explained.

The first example is **themis_dynatable**. In this first example, the secret is not any of the arguments passed to the method but instead is the argument passed to the constructor of the object that calls the vulnerable method. However, for the tool to be able to correct the method, its original driver was changed so that both arguments of the method are considered as secrets. That means that the method has an early-exit timing side-channel vulnerability since there are two exit points of the method that depend on the secret which is also true considering the secret passed to the constructor of the calling object which would be the variable *people*. And considering that a variable used in the stopping condition of the cycle is a secret since its value is dependent on the secrets, the method also has a control-flow based timing side-channel vulnerability. Meaning that this example has a mixed timing side-channel vulnerability. Which means that to correct it, the tool must replace all early return statements for an assignment, ensure that any variable that is being protected by that exit is still protected, and needs to update the stopping condition of the cycle to not use a secret. As seen in the corrected version of the vulnerable method, available in listing 5.10, the tool corrected the early-exit timing side-channel vulnerability but was not able to repair the control-flow. This happens because to correct the control-flow the tool would replace the secret used with the stopping condition with a public argument. However, since both arguments of the method are considered secrets it has no arguments to chose from. So, it can not modify the stopping condition of the cycle, which means it can not repair the vulnerable method.

```
public Person[] getPeople_unsafe$Modification(int startIndex, int maxCount) {
    int $4 = 0;
    int $3 = startIndex;
    Person[] $1;
    Person[] results;
    int peopleCount = people.size();
    int $2 = Math.min(startIndex + maxCount, peopleCount);
    int start = startIndex;
    if (start >= peopleCount) {
        results = NO_PEOPLE;
    } else {
        $1 = NO_PEOPLE;
    }
    int end = Math.min(startIndex + maxCount, peopleCount);
    if (start == end) {
        results = NO_PEOPLE;
    } else {
        $1 = NO_PEOPLE;
    }
    int resultCount = 0;
    if ((!(start >= peopleCount)) && (!(start == end))) {
        resultCount = end - start;
    } else {
        $4 = $2 - $3;
    }
    results = new Person[resultCount];
    for (int from = start, to = 0; to < resultCount; ++from , ++to) {
        results[to] = people.get(from);
    }
    return results;
}
```

Listing 5.10: Vulnerable method getPeople

The next examples are **themis_orientdb** and **themis_picketbox**. Since the vulnerable method is very similar in both examples, only the corrected version of **themis_orientdb** is available in listing 5.11. In this example, the vulnerable method has a mixed timing side-channel vulnerability. As such, the first task of the tool is to ensure the method only contains a final return statement that returns the result of the method execution, which successfully does. Then, it needs to ensure that any branch wherein the condition is a secret regardless of the value of the secret the method execution time is similar. It also needs to guarantee that there is no cycle in the method where the stopping condition does not depend on a secret. Despite improving the method in those regards, the tool was not able to successfully ensure all those requirements. The tool was not able to replace the secret used in the stopping condition of the cycle because there is no public argument with the same type to replace the secret. Although the tool reduces the branches execution time difference depending on the value of the secret, the difference is still not small enough, since some instructions used in the 'then' block of the 'if' statement can't be used in the 'else' block since those instructions use variables used in the condition of the 'if' statement.

```
public boolean equals_inline$Modification(String iPassword, String iHash) {
    boolean $4 = false;
    boolean $1 = false;
    int n = iPassword.length();
    int $5 = n;
    if (n == iHash.length()) {
        char[] v1 = iPassword.toCharArray();
        char[] v2 = iHash.toCharArray();
        int i = 0;
        $1 = true;
        while ((n--) != 0) {
            if (((i < v1.length) && (i < v2.length)) && (v1[i] != v2[i])) {
                $1 = false;
            } else {
                $4 = false;
            }
            i++;
        }
    } else {
        char[] $2 = iPassword.toCharArray();
        int $3 = 0;
        $4 = true;
        while (($5--) != 0) {
            $3++;
        }
    }
    return $1;
}
```

Listing 5.11: Vulnerable method equals_inline

The next example is **themis_spring-security**. In this example, the tool needs to ensure that only one return statement exists in the method and doing so needs to guarantee that all access done to variables used as conditions for the early-exit remain protected. The tool also needs to replace the secret used in the stopping condition of the cycle for a public argument. As seen in listing 5.12, the tool successfully repaired the early-exit but not the control-flow based timing side-channel in the method. This happens because the tool does not know a public argument with the same type as the secret used in the stopping condition of the method.

```
static boolean equals_unsafe$Modification(String expected, String actual) {
    int $3 = 0;
    boolean $2;
    boolean $1;
    byte[] expectedBytes = bytesUtf8(expected);
    byte[] actualBytes = bytesUtf8(actual);
    int expectedLength = (expectedBytes == null) ? -1 : expectedBytes.length;
    int actualLength = (actualBytes == null) ? -1 : actualBytes.length;
    if (expectedLength != actualLength) {
        $1 = false;
    } else {
        $2 = false;
    }
    int result = 0;
    for (int i = 0; i < expectedLength; i++) {
        if ((i < expectedLength) && (i < actualLength)) {
            result |= expectedBytes[i] ^ actualBytes[i];
        } else {
            $3 |= 0;
        }
    }
    $1 = result == 0;
    return $1;
}
```

Listing 5.12: Vulnerable method equals

Next is **themis_tomcat**. This example is a case of a mixed timing side-channel vulnerability. As seen in listing 5.13, the tool repairs the early-exit timing side-channel vulnerability of the method, however when the argument *username*, which is the secret is invalid the result of the call to the method *getPassword* will be null, making the execution of the method *matches* faster than if the secret was valid. As such, to correct this vulnerability, according to the safe version of the method, when the result of the call to the method *getPassword* is null, slow instructions should be executed to 'waste' time, to slow the execution of the method to pretend the secret was valid.

```
protected Boolean authenticate_unsafe$Modification(Connection dbConnection,
    String username, String credentials) {
    Boolean $2 = true;
    Boolean $1 = true;
    if ((username == null) || (credentials == null)) {
        $1 = null;
    } else {
        $2 = null;
    }
    System.out.println("Looking up the user's credentials ...");
    String dbCredentials = getPassword(dbConnection, username);
    if (dbCredentials == null) {
        System.out.println("User not found ...");
        $1 = false;
    } else {
        System.out.println("User not found ...");
        $2 = false;
    }
    boolean validated = matches(credentials, dbCredentials);
    if (!validated) {
        System.out.println("User not validated...");
        $1 = false;
    } else {
        System.out.println("User not validated...");
        $2 = false;
    }
    System.out.println("User is validated...");
    return $1;
}
```

Listing 5.13: Vulnerable method authenticate

Another example is **blazer_gpt14**, visible in listing 5.14. In this example, a problem is that a variable is only considered a secret after the tool analysis the instruction where that variable was used in a condition and this problem is more evident when executing the method because it happens inside a loop. Inside the 'for' loop, there is another 'for' loop and inside this one there is an 'if' statement wherein its condition is used the variable *m*. When the tool analysis this instruction it does not consider to be a possible source of a timing side-channel vulnerability because neither the variable *m* nor the variable *j* is a secret, only the variables *b* and *n* are secrets. However, at the end of the outer loop, the variable *m* is considered a secret because in the final 'if' statement, depending on the value of *b*, a secret, the value of *m* can be altered. Since this happens inside a loop, it affects its execution in subsequent iterations. However, since the tool analysis the method in one passage through the code, it does not notice this possible source of vulnerability. A way to correct this would be to modify the tool so that when it finds a loop it does a first analysis of the body of the loop to update the list of secret variables. Then, it would analyse the loop body again, but this time in search of possible sources for timing side-channel vulnerability with the correct list of secrets.

```
public static BigInteger modular_exponentiation_inline_unsafe$Modification(
     BigInteger a, BigInteger b, BigInteger p) {
    BigInteger $1 = BigInteger.valueOf(1);
    BigInteger m = BigInteger.valueOf(1);
    int n = b.bitLength();
    for (int i = 0; i < n; i++) {
        BigInteger p1;
```

```java
        if (m.testBit(0)) {
            p1 = m;
        } else {
            p1 = BigInteger.valueOf(0);
        }
        int n1 = m.bitLength();
        for (int j = 1; j < n1; j++) {
            if (m.testBit(j)) {
                p = p.add(m);
            }
        }
        BigInteger t = m.multiply(a).mod(p);
        if (b.testBit(i)) {
            m = t;
        } else {
            $1 = t;
        }
    }
    return a;
}
```

Listing 5.14: Vulnerable method of example gpt14

Next is the example **themis_boot-stateless-auth**, that can be seen in listing 5.15, where both parameters are secrets. In this example, the problem creates a slight increase in execution time depending on the value of the secret, one that is hard to explore but exist nevertheless. In the ‘if’ statement of line 21 the variable *a2* is checked for null and then its property length is compared to the variable length. However if the variable *a2* is null, the operator && is short-circuited and the comparison is faster. This then happens again in the ‘if’ condition of line 27. However, this time the effects are greater because not only are there more conditions to validate but it happens inside a loop, so the short-circuits happens several times.

```java
1  public static boolean unsafe_isEqual$Modification(byte[] a, byte[] a2) {
2      int $3 = 0;
3      boolean $2 = true;
4      boolean $1 = true;
5      if (a == a2) {
6          $1 = true;
```

```
    } else {
        $2 = true;
    }
    if ((a == null) || (a2 == null)) {
        $1 = false;
    } else {
        $2 = false;
    }
    int length = 0;
    if (a != null) {
        length = a.length;
    } else {
        $3 = 0;
    }
    if ((a2 != null) && (a2.length != length)) {
        $1 = false;
    } else {
        $2 = false;
    }
    for (int i = 0; i < length; i++) {
        if ((((a != null) && (i < a.length)) && ((a2 != null) && (i < a2.length))
            ) && (a[i] != a2[i])) {
            $1 = false;
        } else {
            $2 = false;
        }
    }
    return $1;
}
```

Listing 5.15: Vulnerable method of example themis_boot-tateless-auth

Finally is the example **stac_ibasys**, that can be seen in listing 5.16. In this example, the problem with correction is connected to possible optimizations of the compiler. In lines 40 and 41, it is passed as an argument to the method *Math.abs* the value stored in the position indicated by *var16* of the arrays *imagedata* and *pcode* respectively. Since the secret in this method is *pcode*, the variable *var16* is influenced by the secret and thus is considered a secret, these instructions are influenced by the secret. However, the lines 50 and 51 that compensate these instructions it is always accessed the arrays *imagedata* and *pcode* in index 0, and this can be optimized by the compiler to be faster since it will access the same value.

```java
public static void test$Modification(byte[] i, byte[] pcode) {
        byte[] $3 = new byte[1];
        boolean $2 = false;
        byte[] imagedata = null;
        boolean success = false;
        boolean state = false;
        try {
            System.out.println("Loading passcode");
            ScalrApplyTest b = new ScalrApplyTest();
            ScalrApplyTest.setup(i);
            BufferedImage p = b.testApply1();
            int r = p.getWidth();
            int h = p.getHeight();
            int[] imageDataBuff = p.getRGB(0, 0, r, h, ((int[]) (null)), 0, r);
            ByteBuffer byteBuffer = ByteBuffer.allocate(imageDataBuff.length * 4)
                ;
            IntBuffer intBuffer = byteBuffer.asIntBuffer();
            intBuffer.put(imageDataBuff);
            ByteArrayOutputStream baos = new ByteArrayOutputStream();
            baos.write(byteBuffer.array());
            baos.flush();
            baos.close();
            System.out.println("Image Done");
            ScalrApplyTest.tearDown();
            byte[] pcodetest = new byte[128];
            int csize = imageDataBuff.length / 128;
            int ii = 0;
            for (int i1 = 0; i1 < (csize * 128); i1 += csize) {
                pcodetest[ii] = ((byte) (imageDataBuff[i1] % 2));
                ++ii;
            }
            imagedata = pcodetest;
            state = true;
        } catch (Exception var15) {
            System.out.println("worker ended, error: " + var15.getMessage());
        }
        if (state) {
            success = true;
            for (int var16 = 0; (var16 < imagedata.length) && (var16 < i.length); var16 += 4) {
                if ((var16 < imagedata.length) && (var16 < pcode.length)) {
                    int var17 = Math.abs(imagedata[var16]);
                    int var18 = Math.abs(pcode[var16]);
                    boolean var19 = (var18 % 2) == (var17 % 2);
                    if (!var19) {
                        success = false;
                    } else {
                        $2 = false;
                    }
                    imagedata[var16] = ((byte) ((var19) ? 1 : 0));
                } else {
                    int var17 = Math.abs(imagedata[0]);
                    int var18 = Math.abs(pcode[0]);
                    boolean $1 = (var18 % 2) == (var17 % 2);
                    if ($1) {
                        $2 = false;
                    } else {
                        $2 = false;
                    }
                    $3[0] = ((byte) (($1) ? 1 : 0));
                }
            }
            System.out.println(" - status:" + success);
        } else {
            success = false;
        }
}
```

Listing 5.16: Vulnerable method of example stac_ibasys

6

Conclusion

Contents

6.1 Conclusions 91
6.2 System Limitations and Future Work 91

6.1 Conclusions

The main goal of this project was to develop a tool for automatic repair of timing side-channel vulnerabilities in Java code. The project aimed at working in conjunction with DifFuzz [1] (for example, DifFuzz's already defined "drivers" were used to identify the vulnerable methods). During the development of this project, patterns and any sort of computation that lead to timing side-channel vulnerabilities were identified. Moreover, algorithms capable of correcting potential timing side-channel vulnerabilities were proposed and implemented.

The tool developed was evaluated using the same dataset that was used to evaluate DifFuzz [1], a dataset that contains examples of applications with timing side-channel vulnerabilities. The results obtained show that 88% of the attempted corrections are semantically correct and 56% eliminate the existing timing side-channel vulnerabilities.

Even though the results show that the tool can improve, can be used as a starting line for the development of new and improved tools capable of correcting timing side-channel vulnerabilities and any other type of vulnerability. For that reason, it is believed that the objectives of this project were met since it lays the groundwork for new and better tools.

The tool is open-source and is available at:

https://github.com/RuiDTLima/DifFuzzAR

6.2 System Limitations and Future Work

Although this tool was built in an attempt to correct timing side-channel vulnerabilities regardless of how they present themselves it is still possible that sometimes the repair created by the tool not only does not repair the vulnerabilities but breaks some of the functionality of the method. As such, it is important to do a manual analysis of the repaired method after the execution of the tool, not only to check if no functionality is broken but also to beautify the changes, like the names of the variables. Besides that, it is important that after the execution of the tool, the produced code is analysed again with DifFuzz to see if the tool eliminated the vulnerability.

The tool assumes that the method referenced in the Driver is vulnerable and corrects it. As such, if the Driver is not properly written or the method referenced is not the vulnerable one, but one that calls the truly vulnerable method, then the tool will not be able to repair it.

Although the repair of a vulnerability is performed automatically, the tool needs to know how to repair the vulnerability given a statement found. For instance, the tool needs to know how to correct a control-flow based timing side-channel vulnerability, when it finds an 'if' statement. This information needs to be added to the tool. As such, it is necessary to continuously improve the tool to be able to correct different code patterns that contain a vulnerability, or different instructions that cause the vulnerability.

If the tool is executed on the correction of a control-flow based timing side-channel vulnerability, it will always try to repair the vulnerability again, which means it might break the original correction.

In the results presented in this report, the corrected version of some examples is presented as having no timing side-channel vulnerability. However, they might have a vulnerability but in the results obtained that vulnerability was not noticed. Despite this, the work developed for this report is open to others on GitHub.

Despite the work performed for the development of this tool, there is still plenty of work that can be done to improve the tool or even to develop a new and better tool.

Future work. One of the most important future directions for the tool is to add the ability to repair more examples of timing side-channel vulnerabilities and to add the ability to repair other types of vulnerabilities so that it can become a one-size-fits-all type of tool.

This tool can only be used after the use of DifFuzz. This means that the user must first use DifFuzz to verify if the application has any timing side-channel vulnerability. Then, the user must use DifFuzz again, but this time to pinpoint the specific method that is vulnerable. Only then, can the user feed the last Driver to this tool, to correct the vulnerability. If the application has several methods with timing side-channel vulnerabilities, this process must be repeated until all instances of vulnerability are repaired. So, to spare the user of this trouble, a future improvement is to integrate the tool with DifFuzz. This way, the user simply provides the new tool with a Driver to the application entry point, and then it searches for every method with a timing side-channel vulnerability. This would mean that a new functionality had to be created so that the tool could generate a Driver automatically.

Another future improvement for the tool is to transform it from a tool into a plugin to be used in the build process of the application. This would reduce the amount of manual intervention needed by the user. Another advantage of this is that being part of the build process can make it easier for other users to use the tool.

Bibliography

[1] S. Nilizadeh, Y. Noller, and C. S. Păsăreanu, “Diffuzz: differential fuzzing for side-channel analysis,” in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 176–187.

[2] T. Antonopoulos, P. Gazzillo, M. Hicks, E. Koskinen, T. Terauchi, and S. Wei, “Decomposition instead of self-composition for proving the absence of timing channels,” vol. 52, no. 6. ACM, 2017, pp. 362–375.

[3] J. Chen, Y. Feng, and I. Dillig, “Precise detection of side-channel vulnerabilities using quantitative cartesian hoare logic,” in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 875–890.

[4] (2020) Bug vs Vulnerability: Know Both Your Enemies. Accessed 2020-10-05. [Online]. Available: https://blog.smartdec.net/bug-vs-vulnerability-d6d4dc4068bd

[5] Y. Zhou and D. Feng, “Side-channel attacks: Ten years after its publication and the impacts on cryptographic module security testing.” *IACR Cryptology ePrint Archive*, vol. 2005, p. 388, 2005.

[6] F. Koeune and F.-X. Standaert, “A tutorial on physical security and side-channel attacks,” in *Foundations of Security Analysis and Design III*. Springer, 2005, pp. 78–108.

[7] Nate Lawson. Timing attack in Google Keyczar library. Accessed 2020-08-17. [Online]. Available: https://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library/

[8] IVC Wiki. Xbox 360 Timing Attack. Accessed 2020-08-17. [Online]. Available: https://beta.ivc.no/wiki/index.php/Xbox_360_Timing_Attack

[9] Apache. (2020) FtpServer. Accessed 2020-08-17. [Online]. Available: https://github.com/apache/ftpserver

[10] AuthMe. (2020) AuthMeReloaded. Accessed 2020-08-17. [Online]. Available: https://github.com/AuthMe/AuthMeReloaded

[11] GitHub. (2019) The state of the Octoverse. Accessed 2019-10-07. [Online]. Available: https://octoverse.github.com/projects#languages

[12] Cloud Foundry. (2020) These Are the Top Languages for Enterprise Application Development And What That Means for Busines. Accessed 2020-08-17. [Online]. Available: https://www.cloudfoundry.org/wp-content/uploads/Developer-Language-Report_FINAL.pdf

[13] IBM. (2020) Modern languages for the modern enterprise. Accessed 2020-08-17. [Online]. Available: https://developer.ibm.com/articles/d-modern-language-modern-enterprise/

[14] N. Meng, S. Nagy, D. Yao, W. Zhuang, and G. Arango-Argoty, "Secure coding practices in java: Challenges and vulnerabilities," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 372–383.

[15] DARPA. (2020) Space/Time Analysis for Cybersecurity (STAC). Accessed 2020-08-17. [Online]. Available: https://www.darpa.mil/program/space-time-analysis-for-cybersecurity

[16] J. A. Goguen and J. Meseguer, "Security policies and security models," in *1982 IEEE Symposium on Security and Privacy*. IEEE, 1982, pp. 11–11.

[17] M. Zalewski, "American fuzzy lop," 2017. [Online]. Available: http://lcamtuf.coredump.cx/afl

[18] R. Kersten, K. Luckow, and C. S. Păsăreanu, "Poster: Afl-based fuzzing for java with kelinci," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2017, pp. 2511–2513.

[19] C. Le Goues, M. Pradel, and A. Roychoudhury, "Automated program repair," 2019.

[20] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "Genprog: A generic method for automatic software repair," *Ieee transactions on software engineering*, vol. 38, no. 1, pp. 54–72, 2011.

[21] J. Xuan, M. Martinez, F. Demarco, M. Clement, S. L. Marcote, T. Durieux, D. Le Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, vol. 43, no. 1, pp. 34–55, 2016.

[22] M. Monperrus, "Automatic software repair: a bibliography," *ACM Computing Surveys*, vol. 51, 01 2015.

[23] F. DeMarco, J. Xuan, D. Le Berre, and M. Monperrus, "Automatic repair of buggy if conditions and missing preconditions with smt," in *Proceedings of the 6th international workshop on constraints in software testing, verification, and analysis*. ACM, 2014, pp. 30–39.

[24] M. Wu, S. Guo, P. Schaumont, and C. Wang, “Eliminating timing side-channel leaks using program repair,” in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018, pp. 15–26.

[25] (2020) EvoSuite: Automatic Test Suite Generation for Java. Accessed 2020-08-27. [Online]. Available: https://www.evosuite.org/

[26] (2020) Javassist: Java bytecode engineering toolkit since 1999. Accessed 2020-10-02. [Online]. Available: https://www.javassist.org/

[27] (2020) Spoon - Source Code Analysis and Transformation for Java. Accessed 2020-10-02. [Online]. Available: http://spoon.gforge.inria.fr/