Automatic Repair of Java Code with Timing Side-Channel Vulnerabilities

Rui Lima
Universidade de Lisboa - Instituto Superior Técnico
Lisboa, Portugal
rui.tomas.lima@tecnico.ulisboa.pt

***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.**

***Index Terms*—Java, Timing Side-Channel Vulnerabilities, Automatic Repair of Vulnerabilities, Security, Code Modification**

I. Introduction

Software is increasingly more present in the world and our lives. Virtually all industries today use software in one way or another. It is thus important that software is 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.

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 [1]. 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 [2], [3]. 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. Moreover, 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 [4]. Another example is the timing side-channel vulnerability discovered in Xbox 360 [5].

As timing side-channel vulnerabilities are extremely difficult to detect, there is a great effort by researchers to develop tools capable of automatically detecting these vulnerabilities [6]–[8]. 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 [9], Java is the second language with more contributors in public and private repositories and is still the most used language for enterprise applications [10], [11]. The tool developed, called *DifFuzzAR*, is designed to work in conjunction with DifFuzz [8] 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.

II. Background and Related Work

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. [12] 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 related to authentication, cryptography, Java EE security, access control, and secure communication.

A. 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. 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 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.

1) Early-Exit Vulnerabilities: An early-exit timing side-channel vulnerability happens when the method contains exit points where their execution depends on the value of a secret. This is very common when checking if an array has a certain length, if not then the execution of the method ends. If that array is a secret, then that means that the execution time of the method is dependent on the value of the secret, in this case, on the size of the array.

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.

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 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.

B. Automated Detection of Timing Side-Channel Vulnerabilities

Automatic detection of timing side-channel vulnerabilities has received substantial attention in recent years.

In 2017, Timos Antonopoulos et al. [6] developed a new way to prove the absence of timing side-channels. They used decomposition instead of self-composition to prove the absence of timing side-channels. 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.

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. Their method is implemented in a tool called Blazer.

Jia Chen et al. [7] presented the notion of ϵ-bounded non-interference, a variation of Goguen and Meseguer's non-interference principle [13]. 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 denote 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.

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 QCHL 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*". The authors implemented their technique in a tool

[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.

called Themis and showed that their tool can find previously unknown vulnerabilities in widely used Java programs.

In 2019, Shirin Nilizadeh et al. [8] 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*". 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, they use the fuzz testing tool *American Fuzzy Lop* (AFL) [14], which uses genetic algorithms and mutates 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 [15] 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. And this process is repeated for a predetermined time or until the user cancels the execution of the tool.

To assess the correctness of DifFuzz, the authors applied 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 [16] 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.

C. Automated Repair Tools

In 2012, Claire Le Goues et al. [17] presented a generic method for automatic software repair called GenProg, which 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 cases encodes the set of functionality that can not be lost while repairing the bug. 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. 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.

In 2017 Jifeng Xuan et al. [18] 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 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 that is used to rank statements in a descending order based on their suspiciousness score. The suspiciousness score indicates the likelihood that a statement contains a bug. Each 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. To evaluate Nopol, the authors executed it on a dataset with 22 real-world bugs from Apache Commons Math and Apache Commons Lang. 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.

D. Automated Repair of Timing Side-Channel Vulnerabilities

Meng Wu et al. [19] 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

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

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.

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.

III. 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.

An overview of the architecture of the tool is shown in Figure 1.

IV. 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 instructions 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 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 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,

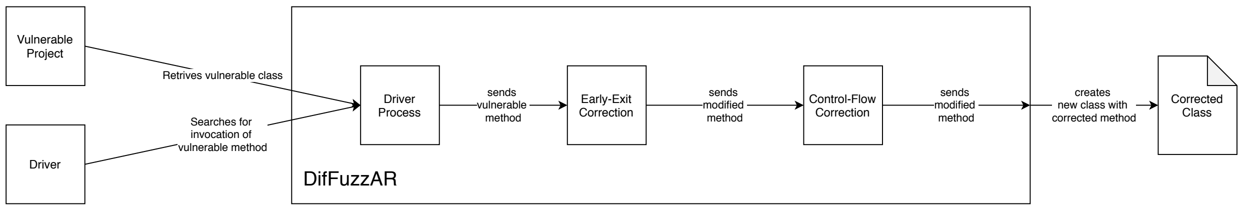

Fig. 1. Overview of DifFuzzAR

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 1. 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 1: 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)
```

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.

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.

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). 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.

V. 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 2 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 2: Overview of the repair process

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

A. Correction of Early-Exit Timing Side-Channel Vulnerabilities

The correction of early-exit timing side-channel vulnerabilities consists in 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.

Algorithm 3 shows 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 analyses 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, 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 *VulnerableMethodName$Modification*. So, if the users want to use the corrected version, they must replace the original method for the corrected method.

This simple modification can create or exacerbate a control-flow based timing side-channel vulnerability. For instance, if

Algorithm 3: 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()
```

an early-exit happens inside a cycle, where the stopping condition depends on a 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.

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.

B. Correct Control-Flow Timing Side-Channel Vulnerabilities

The correction of control-flow based timing side-channel vulnerabilities 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.

Algorithms 4 and 5 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 4: Correction of Control-Flow Based Timing Side-Channel Vulnerabilities - PART 1

```
1:  secrets ← createListOfSecrets(vulnMethod)
2:  public ← createListOfPublic(vulnMethod)
3:  replacements ← newMap()
4:  instruction ← getNextInstruction(vulnMethod)
5:  while exist(instruction) do
6:    if isAssignment(instruction) then
7:      if valueAssignedUsesSecret(instruction, secrets) then
8:        variable ← getVariableAssignedTo(instruction)
9:        secrets ← addToSecrets(variable, secrets)
10:     end if
11:     if toModify(instruction) then
12:       newVar ← createNewVariable(instruction)
13:       addToReplacements(replacements, instruction, newVar)
14:       changeVariableAssignedTo(instruction, newVar)
15:     end if
16:   else if isForStatement(instruction) then
17:     if conditionUsesSecret(instruction, secrets) then
18:       changeConditionToUsePublic(instruction, secrets, public)
19:     end if
20:     traverseForBody(instruction)
21:   else if isIfStatement(instruction) then
22:     thenBlock ← getThenBlock(instruction)
23:     elseBlock ← getElseBlock(instruction)
24:     if conditionUsesSecret(instruction, secrets) then
25:       modThen ← modifyInstructions(thenBlock)
26:       modElse ← modifyInstructions(elseBlock)
27:       addToStartOfBlock(modThen, thenBlock)
28:       addToStartOfBlock(modElse, elseBlock)
29:     else
30:       traverseBlock(thenBlock)
31:       traverseBlock(elseBlock)
32:     end if
33:   else if isInvocation(instruction) then
34:     target ← getInvocationTarget(instruction)
35:     if isSecret(target, secrets) then
36:       newTarget ← createNewVariable(target)
37:       addToReplacements(replacements, instruction,
    newTarget)
38:       replaceTarget(instruction, newTarget)
39:     end if
```

C. 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.

VI. Evaluation

In this section, 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.

A. 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

Algorithm 5: Correction of Control-Flow Based Timing Side-Channel Vulnerabilities - PART 2

```
    else if isLocalVariable(instruction) then
        assigned ← getValueAssigned(instruction)
        if usesSecret(assigned, secrets) then
            variableAssigned ← getVariableAssignedTo(instruction)
            secrets ← addtoSecrets(variableAssigned, secrets)
        else if !isPartOfConditionOfParentIf(instruction, assigned) then
            newVar ← createNewVariable(instruction)
            addToReplacements(replacements, instruction, newVar)
            changeVariableAssignedTo(instruction, newVar)
        end if
    else if isLoopStatement(instruction) then
        traverseLoopBody(instruction)
    else if isOperatorAssignment(instruction) then
        newVar ← createNewVariable(instruction)
        addToReplacements(replacements, instruction, newVar)
        changeVariableAssignedTo(instruction, newVar)
    else if isTryBlock(instruction) then
        traverseTryBody(instruction)
    else if isUnaryOperator(instruction) then
        var ← getVariable(instruction)
        if isInReplacements(replacements, var) then
            replacement ← getReplacement(replacements, var)
            changeVariableUsed(instruction, replacement)
        end if
    else if isWhileStatement(instruction) then
        condition ← getStoppingCondition(instruction)
        if usesReplacedVariable(condition, replacements) then
            condition ← getReplacement(replacements, instruction)
        else
            condition ← createNewVariable(condition)
            addToReplacements(replacements, instruction, condition)
        end if
        updateStoppingCondition(instruction, condition)
        traverseWhileBody(instruction)
    end if
    instruction ← getNextInstruction(vulnMethod)
end while
newMethod ← saveChanges()
```

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. 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.

B. 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 [20]. 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 considered to be semantically correct.

C. 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 I.

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%.

VII. 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* [8] (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* [8], 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, it can be used as a starting point for the development of new and improved tools capable of correcting timing side-channel vulnerabilities and other related vulnerabilities. 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

A. System Limitations

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 versions of some examples are presented as having no timing side-channel vulnerability. However, there is always the possibility that they might have a vulnerability that remained unnoticed. Despite this, the work developed for this report is open to others on GitHub.

B. Future work

Despite the work performed for the development of this tool, there is still plenty of work that can be done to improve the tool. 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.

References

[1] (2020) Bug vs Vulnerability: Know Both Your Enemies. Accessed 2020-10-05. [Online]. Available: https://blog.smartdec.net/bug-vs-vulnerability-d6d4dc4068bd

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 I
RESULTS OF TOOL

[2] 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.

[3] 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.

[4] 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/

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

[6] 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," *ACM SIGPLAN Notices*, vol. 52, no. 6, pp. 362–375, 2017.

[7] 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.

[8] 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.

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

[10] 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

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

[12] 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.

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

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

[15] 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.

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

[17] 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.

[18] 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.

[19] 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.

[20] (2020) EvoSuite: Automatic Test Suite Generation for Java. Accessed 2020-08-27. [Online]. Available: https://www.evosuite.org/