Automatic Repair of Java Code with Timing Side-Channel Vulnerabilities

Rui Lima and João F. Ferreira
INESC-ID and IST, University of Lisbon, Portugal

Alexandra Mendes
INESC TEC and University of Beira Interior, Portugal

***Abstract*—Vulnerability detection and repair is a demanding and expensive part of the software development process. As such, there has been an effort 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. In this paper, we propose DifFuzzAR, a new tool for automatic repair of timing side-channel vulnerabilities in Java code. The tool works in conjunction with DifFuzz and it is 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 those that are control-flow based.**

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

I. Introduction

Software vulnerabilities are a serious threat to the security of software systems and can have disastrous consequences. For that reason, the detection of software vulnerabilities is an important problem that has received a lot of attention from the software security community. However, the detection of vulnerabilities can be difficult since a vulnerable application can pass all tests or even fulfil its correctness specification. Different types of vulnerabilities have different difficulty levels of detection. Arguably, 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 which can manifest in several different ways: for example, in the difference in computation time, in power consumption, sound production, or electromagnetic radiation emitted [1], [2]. Most of the side-effects require that the attackers have physical access to the system they are trying to attack, since they need to gather information directly from the system (e.g. measuring the power consumption). On the other hand, side-effects such as the difference in computation time or in response size do not require the attackers to be in direct contact with the systems under attack. This enables the possibility of remote attacks, thus exposing 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: for example, 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 [3]; another example is the timing side-channel vulnerability discovered in Xbox 360 [4].

As timing side-channel vulnerabilities are difficult to detect, there has been a substantial effort to develop tools capable of automatically detecting these vulnerabilities [5]–[7]. 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, we propose to facilitate the correction of vulnerabilities by developing a tool capable of automatically repairing timing side-channel vulnerabilities. Even though the ideas presented in this paper are general and can be applied to different programming languages, we focus on Java, since according to GitHub [8], Java is the second language with more contributors in public and private repositories and is still the most used language for enterprise applications [9], [10]. The tool developed, called *DifFuzzAR*, is designed to work in conjunction with the state-of-the-art detection tool DifFuzz [7]. We evaluated *DifFuzzAR* 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.

Structure of the paper: We present background and related work in Section II. After presenting an overview of the system in Section III, we describe the main components of *DifFuzzAR*: in Section IV we present how vulnerable methods are identified and in Section V we describe how vulnerabilities are fixed. The evaluation of the tool is presented in Section VI. We conclude the paper in Section VII, where we also present current limitations and discuss future work.

II. Background and Related Work

This section presents background and related work on timing side-channel vulnerabilities and automated repair methods.

A. 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

[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.

timing side-channel attacks when the time it takes to complete a computation depends on a given secret, e.g., a password. A timing side-channel vulnerability can appear in multiple ways, as detailed below.

1) Early-Exit Vulnerabilities: An *early-exit timing side-channel vulnerability* happens when the method contains exit points that are dependent on the value of a secret. An example is when checking if an array has a certain length and exiting immediately once it is established that it does not. If that array is a secret, then 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 vulnerability* happens when there is a significantly slow operation that happens only when a certain condition is met. This means that an attacker can take notice of the time a method takes to return and learn that the slow 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, we say that the method has a *mixed timing side-channel vulnerability*. To fix it, it is necessary to correct the early-exit and the control-flow parts of the vulnerability. This can be done in different ways. First, they can be corrected simultaneously. 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.

B. Detection of Timing Side-Channel Vulnerabilities

Automated detection of timing side-channel vulnerabilities has received substantial attention in recent years. Timos Antonopoulos et al. [5] developed a new way to prove the absence of timing side-channels, by using decomposition instead of self-composition. 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. The authors 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. [6] presented the notion of ϵ-bounded non-interference, a variation of Goguen and Meseguer's non-interference principle [11]. 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 called Themis and showed that their tool can find previously unknown vulnerabilities in widely used Java programs.

Shirin Nilizadeh et al. [7] present a new approach based on dynamic analysis and introduce a new tool called *DifFuzz* that uses differential fuzzing[2]. *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) [12], which uses genetic algorithms and mutates the inputs using byte-level coverage. Given that AFL only supports programs written in C, C++, or Objective C, and *DifFuzz* is written in Java, the authors used Kelinci [13] to connect the two tools, since Kelinci provides AFL-style instrumentation for Java programs. To use *DifFuzz*, one has to create a Fuzzing Driver (or Driver File), 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 that the difference can be increased. This process is repeated for a predetermined time or until the user cancels the execution of the tool. *DifFuzz* was evaluated with a dataset of widely-used Java applications and it found previously unknown vulnerabilities (later confirmed by the developers). It was also applied to complex examples from the DARPA STAC [14] program. *DifFuzz* was able to find the same vulnerabilities as other tools and also found vulnerabilities on corrected versions of the benchmarks of Themis and Blazer.

The authors of *DifFuzz* proposed improvements such as adding statistical guarantees to the tool and adding automated repair methods to eliminate the vulnerabilities discovered by *DifFuzz*. Our work contributes to the latter.

C. Automated Repair Tools

Claire Le Goues et al. [15] presented a generic method for automatic software repair called GenProg, which receives as input the defected source code and a set of test cases. The set of test cases must contain a set of passing positive test cases and at least one failing negative test case. The negative test case encodes the fault to be repaired and the set of positive test cases encodes the functionalities 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 fault in question.

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

Jifeng Xuan et al. [16] presented Nopol, an 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 input program with a conditional expression. The set of test cases passed as input is similar to the one expected by GenProg. However, 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 a 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 fault.

D. Automated Repair of Timing Side-Channel Vulnerabilities

Meng Wu et al. [17] proposed a method based on program analysis and transformation to eliminate timing side-channel vulnerabilities. Their solution produces a transformed program functionally equivalent to the original program but without instruction and cache timing side-channels. 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 and uses static analysis to identify the set of variables whose values depend on the secret inputs. 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. The method is implemented in a tool called SC-Eliminator.

III. System Overview

DifFuzzAR is designed to work in conjunction with *DifFuzz*. 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 its current version, *DifFuzzAR* 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 another type of vulnerability, they simply have to create a new independent module with all the code capable of repairing it. 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, which is then 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 outputs a new class with the corrected method. An overview of the architecture of the tool is shown in Figure 1.

IV. Identification of Vulnerable Methods

The first task of *DifFuzzAR* is to uncover the vulnerable method that is to be repaired. As mentioned above, the driver used for *DifFuzz* is used to identify the method. This means that the driver must be properly created so that the correct method is retrieved. We assume that drivers are similar to the drivers provided by *DifFuzz*, where each driver calls the vulnerable method twice, each time immediately after a call to the method $Mem.clear()$[3]. However, there are three groups of variations that we consider:

Group 1. The simplest variation 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 variation 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 variation 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.

The search for the method after the $Mem.clear()$ instruction is done twice since in the driver the vulnerable method will be invoked twice. The parameter that changes in both invocations of the method is considered to be the secret. We thus assume that the driver uses the same arguments for the public parameters and different ones for the secret. For example, consider the method invocations $vulnMethod(a, b, c)$ and $vulnMethod(a, d, c)$; here, the second parameter is considered by the tool as the secret, since it is the only parameter that changes. In the identification of the vulnerable method,

[3]For concrete examples, see https://github.com/sr-lab/DifFuzzAR/tree/master/src/test/resources

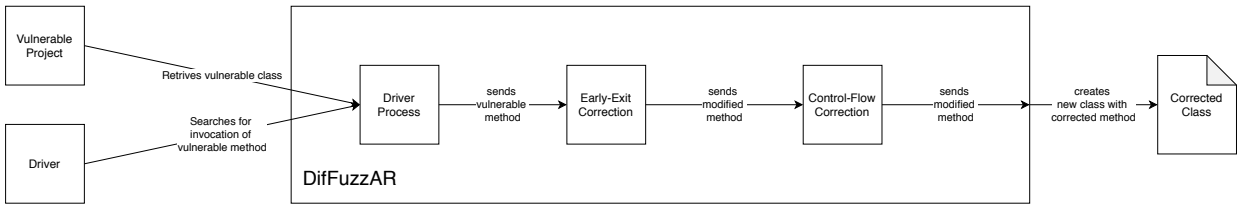

Fig. 1. Overview of DifFuzzAR

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 parameter is different. A basic overview of this process can be seen in Algorithm 1.

Algorithm 1: Identification of the vulnerable method using a *DifFuzz* driver

```
f ← findDriverFile(driverPath)
instMem1, f' ← findMemClear(f)
vulnOpt1 ← recordNextInstruction(instMem1)
instMem2 ← findMemClear(f')
vulnOpt2 ← recordNextInstruction(instMem2)
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 [7]. The tool was capable of finding the correct vulnerable method in all examples.

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 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 potentially have an early-exit and so the tool starts by correcting that vulnerability. Afterwards, the tool executes the module responsible for the correction of control-flow based timing side-channel vulnerabilities.

The tool is implemented in Java and uses the open-source library Spoon [18] for the refactoring process. Examples of corrected vulnerabilities, which can be useful to understand in more detail the descriptions presented in the next subsections, are available in our GitHub repository.[4]

Algorithm 2: Overview of the repair process

```
if numberReturns >1 then
    vulnMethod ← repairEarlyExit(vulnMethod)
end if
vulnMethod ← repairControlFlow(vulnMethod)
```

A. Correcting 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 vulnerabilities. The tool starts by obtaining the element returned in the final return of the method. If this element is not a variable, the tool creates a new variable of the same type as the return type of the method, and initializes it with the element obtained, referred from now on as the return variable. Then, the tool analyses every instruction of the method in search for 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 a copy of the

[4]Examples in GitHub repository: https://github.com/sr-lab/DifFuzzAR/tree/master/src/test/resources

original version, except that it contains an extra method called *VulnerableMethodName$Modification*. If the users want to use the corrected version, they must replace the original method with the corrected method.

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()
```

This correction can create or exacerbate a control-flow based timing side-channel vulnerability. For instance, if an early-exit happens inside a loop, 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.

B. Correcting Control-Flow Timing Side-Channel Vulnerabilities

The correction of control-flow based timing side-channel vulnerabilities involves 1) the modification of the stopping condition of loops that depend on a secret to depend on a public argument or 2) 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. The pseudo-code is divided into two parts to improve presentation. In this process, the tool starts by creating a list of the secrets and a list 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* that needs to be modified, then a new variable is created and it is added to the map of replacements with the existing variable. The instruction is also 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 a *method 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 loop and will analyse its instructions.

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

C. Correcting Mixed Timing Side-Channel Vulnerabilities

Sometimes a method has both an early-exit and a control-flow based timing side-channel vulnerability. 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 its final version. This means that each module responsible for correcting a type

Algorithm 4: 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 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()
```

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 not have two variables with the same name, the naming of a variable is global to the tool and it keeps a record of the names used.

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 timing side-channel vulnerabilities detected by *DifFuzz*. This section presents both types of evaluation, explaining how they are done as well as why they are necessary.

This evaluation 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. *DifFuzz* was configured to run for 2.5 hours. The results of the evaluation can be seen in Table I.

A. Dataset Used

We started with the 32 examples distributed with *DifFuzz*. One of the examples suffers from a size side-channel. In other examples the vulnerability does not follow the template of vulnerable methods considered in this project. Also, it was not possible to understand why some examples are vulnerable. As such, 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; the remaining 15 examples have a mixed timing side-channel vulnerability.

B. Semantics Preservation

The modifications proposed can 'break' the code, in the sense that for the same inputs, the output can be different from that of the original version. As such, it is important that after any modifications to a method, the method is tested again to ensure that its functionality remains.

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 do not include tests, so it was not possible to ensure that the correction kept the functionality of the method. Since creating manual tests is a time-consuming and error-prone activity, we decided to use EvoSuite [19] to generate tests automatically. The tests were created and first run on the original, vulnerable, code. We only retained tests that pass. Then, the vulnerable method was replaced with the method created by the tool and the tests were executed again. If all retained tests passed, then the solution created by the tool to correct the timing side-channel vulnerability was considered to be *semantically correct*.

Table I shows that 22 of the 25 attempted corrections (88%) are semantically correct. Regarding the 3 corrections that are not semantically correct, the first of them fails at compile time because the correction introduced a new variable that was used before being declared; the remaining two fail because, when removing a return to deal with the early-exit vulnerability, an exception *ArrayIndexOutOfBoundsException* is introduced.

C. Vulnerability Correction

Once the tool repairs a vulnerable method and that repair is shown to be semantically correct, it is necessary to verify if the repair produced by the tool repairs the vulnerability. We use *DifFuzz* to determine if the repaired version contains any timing side-channel vulnerabilities.

Table I shows that out of 25 examples, the tool successfully corrected 14 of them, a success rate of 56%. Not all corrected versions produced by the tool are semantically correct, meaning that the code lost some of the functionality after the repair. When considering only semantically correct examples, the total of examples is 22, which makes a success rate of 63,6%.

VII. Conclusions

This paper presents a tool for automatic repair of timing side-channel vulnerabilities in Java code that works in conjunction with *DifFuzz* [7]. Patterns that lead to timing side-channel vulnerabilities were identified and algorithms capable of correcting those potential vulnerabilities were proposed and implemented. The tool developed was evaluated using the same dataset that was used to evaluate *DifFuzz* [7], 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 (i.e. the original behavior is preserved) and 56% of the corrections eliminate the existing timing side-channel vulnerabilities. Even though there is space for improvement, we believe that *DifFuzzAR* 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. The tool is open-source and is available at: https://github.com/sr-lab/DifFuzzAR

A. System Limitations

Although *DifFuzzAR* 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 also 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 (e.g. improve 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.

DifFuzzAR can automatically repair the patterns identified and described in this paper. For vulnerabilities that follow other types of patterns, the tool needs to be extended. It is thus 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 paper, 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, all the work developed is open to others on GitHub.

B. Future work

There is still plenty of work that can be done to improve *DifFuzzAR*. An important direction is to add the ability to repair more examples of timing side-channel vulnerabilities (including patterns not considered).

DifFuzzAR is designed to be used in conjunction with *DifFuzz*. This means that the user must create a Driver following the rules described in Section IV. A future direction that would greatly simplify the use of the tool is to automatically generate a Driver file. 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. Another direction would be to adapt *DifFuzzAR* so that it could easily be used and distributed as an IDE plugin. For this, it is likely that adjustments to the code transformations (e.g. better variable names) and usability studies should be performed, to ensure that the code transformations are accepted by the programmers. It would also be interesting to apply the tool to public projects and

TABLE I
RESULTS OF THE APPLICATION OF *DifFuzzAR* TO THE *DifFuzz* DATASET

Example 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 dependent)	Yes	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 Modpow1	Yes	Control-Flow	Yes	Yes	Yes
Blazer PasswordEq	Yes	Early-Exit (If dependent)	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 Ibasys	No	Control-Flow	Yes	Yes	No
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 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

submit any corrections found as pull requests, thus improving existing software and, simultaneously, obtaining code reviews from developers (as done in related refactoring projects [20]).

ACKNOWLEDGEMENTS

We thank the anonymous reviewers for their valuable and constructive comments. This work was partially funded by the PassCert project, a CMU Portugal Exploratory Project funded by Fundação para a Ciência e Tecnologia (FCT), with reference CMU/TIC/0006/2019 and supported by national funds through FCT under project UIDB/50021/2020.

REFERENCES

[1] 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.

[2] 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.

[3] 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/

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

[5] 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.

[6] 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.

[7] 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.

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

[9] 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

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

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

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

[13] 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.

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

[15] 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.

[16] 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.

[17] 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.

[18] (2020) Spoon - Source Code Analysis and Transformation for Java. Accessed 2020-10-02. [Online]. Available: http://spoon.gforge.inria.fr/

[19] (2020) EvoSuite: Automatic Test Suite Generation for Java. Accessed 2020-08-27. [Online]. Available: https://www.evosuite.org/

[20] A. Ribeiro, J. F. Ferreira, and A. Mendes, "EcoAndroid: An Android Studio plugin for developing energy-efficient Java mobile applications," 2021.