Automated Software Engineering (2024) 31:1
https://doi.org/10.1007/s10515-023-00398-6

DifFuzzAR: automatic repair of timing side-channel vulnerabilities via refactoring

Rui Lima[1,2] · **João F. Ferreira**[1,2] · **Alexandra Mendes**[3,4] · **Carolina Carreira**[1,2]

Received: 20 August 2022 / Accepted: 4 September 2023

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 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 automatically correct timing side-channel vulnerabilities, being more effective with those that are control-flow based. In addition, the results of a user study show that users generally trust the refactorings produced by DifFuzzAR and that they see value in such a tool, in particular for more critical code.

Keywords Source code refactoring · Timing side-channel vulnerabilities · Automatic repair of vulnerabilities · Code repair · Security · Java

1 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

✉ João F. Ferreira
joao@joaoff.com

[1] INESC-ID, Lisbon, Portugal

[2] IST, University of Lisbon, Lisbon, Portugal

[3] HASLab/INESC TEC, Porto, Portugal

[4] Faculty of Engineering, University of Porto, Porto, Portugal

Published online: 18 October 2023

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 (Zhou and Feng 2005; Koeune and Standaert 2005). 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 (Lawson 2020); another example is the timing side-channel vulnerability discovered in Xbox 360 (IVC Wiki 2020).

As timing side-channel vulnerabilities are difficult to detect, there has been a substantial effort to develop tools capable of automatically detecting these vulnerabilities (Antonopoulos et al. 2017; Chen et al. 2017; Nilizadeh et al. 2019). 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 (2019), Java is the second language with more contributors in public and private repositories and is still the most used language for enterprise applications (Cloud Foundry 2020; IBM 2020). The tool developed, called *DifFuzzAR*, is designed to work in conjunction with the state-of-the-art detection tool DifFuzz Nilizadeh et al. (2019). 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. We also performed a quantitative and subjective user study to gather insights on whether potential users would like to use *DifFuzzAR* and if they trust its results, and to collect suggestions for improvement. The results are positive and show that *DifFuzzAR* has the potential to be used and trusted by its target users and that these do see value in the tool, in particular for more critical code.

This paper is an extension of the work originally presented in Lima et al. (2021). In particular, we improve the presentation, we clarify descriptions, we extend parts

of the related work, and we add new examples of refactorings that aid understanding. We also substantially extend the evaluation section by describing a new user study. In particular, we address the three following research questions:

RQ1. Do users trust DifFuzzAR's refactorings?

RQ2. What would users like to see in a tool such as DifFuzzAR?

RQ3. Do users value the use of DifFuzzAR differently in critical and non-critical applications?

1.1 Structure of the paper

We present background and related work in Sect. 2. After presenting an overview of the system in Sect. 3, we describe the main components of *DifFuzzAR*. In Sect. 4 we present how vulnerable methods are identified and in Sect. 5 we describe how vulnerabilities are fixed. The evaluation of the tool is presented in Sect. 6. We discuss threats to validity in Sect. 7 and we conclude the paper in Sect. 8, where we also present current limitations and discuss future work.

2 Background and related work

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

2.1 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. These attacks are practical and apply to general software systems. For example, in their influential work, Brumley and Boneh were able to extract private keys from an OpenSSL-based web server running on a machine in the local network (Brumley and Boneh 2005). As mentioned in the previous section, timing side-channel vulnerabilities were found in Google's Keyczar Library (Lawson 2020) and Xbox 360 (IVC Wiki 2020). A timing side-channel vulnerability can appear in multiple ways, as detailed below.

[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.

2.1.1 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.

Listing 1 shows a classic example of an early-exit vulnerability, consisting of code that checks if the given password is correct. In this code, an exit condition is dependent on the secret, the parameter `sec`. Resourceful attackers can find the password by trying different combinations of characters until they find the correct one, as they know they found a correct character simply by the time the function takes to respond. For example, if the password is "bcdef" and the attackers use the password "abcde", the function returns false immediately. If they use the password "bbcde", they will know that the first character of the password is correct simply because the function took extra clock cycles to return. With that, the attackers know that the first character of the password is "b". They can now repeat the process for the other characters of the password.

```
1  boolean pwcheck_unsafe(byte[] pub, byte[] sec) {
2      if(pub.length != sec.length) {
3          return false;
4      }
5      for(int i = 0; i < pub.length; i++) {
6          if(pub[i] != sec[i]) {
7              return false;
8          }
9      }
10     return true;
11 }
```

Listing 1 Early-Exit Vulnerable Code Snippet [7]

2.1.2 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.

In the example shown in Listing 2, if the condition in line 7 is true, two slow methods will be executed, while if the condition is false, the execution time will be much faster. In this case, it is possible to learn information about the `exponent` through the timing behavior of the program, since the value of the `exponent` influences the execution time.

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

Listing 2 Example of branch dependent execution time [6]

2.1.3 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. As an example, Listing 3 shows a method with a mixed timing side-channel vulnerability, where the parameter `a` is a secret.

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

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

Listing 3 Example of a Mixed timing side-channel vulnerability in a method [7]

2.2 Detection of timing side-channel vulnerabilities

Automated detection of timing side-channel vulnerabilities has received substantial attention in recent years. Antonopoulos et al. (2017) 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.

Chen et al. (2017) presented the notion of ϵ-bounded non-interference, a variation of Goguen and Meseguer's non-interference principle (Goguen and Meseguer 1982). 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.

These two works improve the field of detection of side-channel vulnerabilities. However, both use static analysis.

Nilizadeh et al. (2019) 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) (Zalewski 2017), 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 (Kersten et al. 2017) 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.

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

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 (2020) 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.

2.3 Automated repair tools

Automated program repair consists in fixing software bugs with little to no human intervention. Usually, workflows for automated program repair start with fault localization, which aims at locating faults in the code; patch generation, which aims to generate valid patches; and patch validation, which checks the validity of the generated patches in the previous step. Validity of patches is often determined by checking whether the patched program passes a given test suite. As described in Goues et al. (2019), automated program repair techniques can be classified as *heuristic-based* (Forrest et al. 2009; Kim et al. 2013; Liu et al. 2019; Cornu et al. 2015), *constraint-based* (Nguyen et al. 2013; Xuan et al. 2016; Mechtaev and Roychoudhury 2016), and *learning-based* (Gupta et al. 2017; Chen et al. 2019; Lutellier et al. 2020; Ye et al. 2021; Chen et al. 2021; Yasunaga and Liang 2021; Allamanis et al. 2021; Yasunaga and Liang 2020).

Two influential works in the area are GenProg (Le Goues et al. 2011) and Nopol (Xuan et al. 2016). GenProg receives as input the faulty 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.

Xuan et al. (2016) 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.

For a comprehensive survey on automated program repair, we recommend Monperrus's survey (Monperrus 2015). Moreover, since this is a fast-moving research

topic with new approaches being regularly proposed, we also recommend Monperrus's *living review* on automated program repair (Monperrus 2018).

2.4 Automated repair of timing side-channel vulnerabilities

Wu et al. (2018) 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.

3 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, Control-Flow Based Timing Side-Channel vulnerabilities, and, by combining both, Mixed 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 Fig. 1.

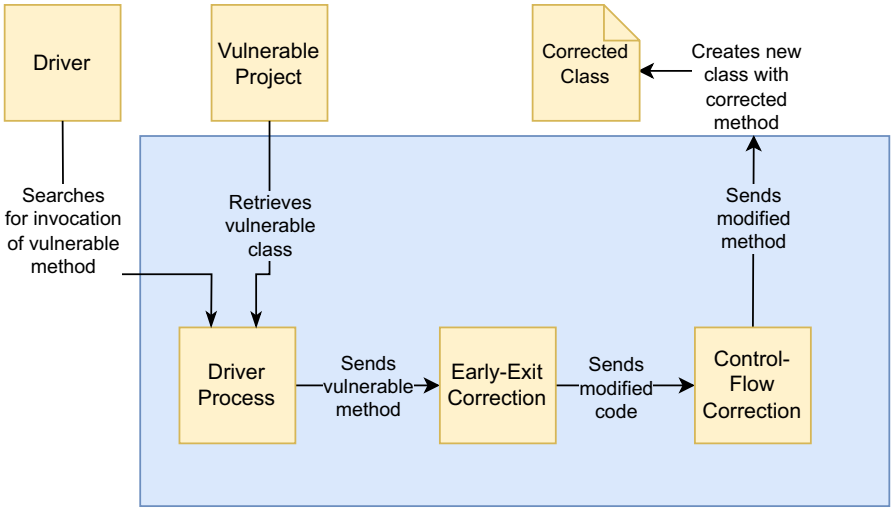

Fig. 1 Overview of DifFuzzAR

4 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.

[3] *DifFuzz* instruments executions and keeps track of execution costs (so that it can detect discrepancies between executions). The method *Mem.clear*() resets the value of the current cost, which is necessary to measure the cost for each execution separately. For concrete examples, see https://github.com/sr-lab/DifFuzzAR/tree/master/src/test/resources.

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. Following *DiffFuzz*'s convention, 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, 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 (Nilizadeh et al. 2019). The tool was capable of finding the correct vulnerable method in all examples.

5 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 (Pawlak et al. 2015) 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

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

5.1 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

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

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 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

```
returnElem ← obtainElemReturnedMethod(vulnMethod)
if !isVariable(returnElemen) then
    returnElem ← createVariable(returnElem)
end if
instruction ← getNextInstruction(vulnMethod)
while exist(instruction) do
    if isReturn(instruction) then
        if afterWhileBlock(instruction) then
            addAssignmentBeforeWhile(instruction, returnElem)
        else if lastReturn(instruction) then
            changeReturnElem(instruction, returnElem)
        else
            replaceWithAssignment(instruction, returnElem)
            if insideIf(instruction) then
                condition ← saveCondition(instruction)
            end if
        end if
    else if isVariableProtected(instruction) then
        addIfToVariable(instruction)
    end if
    instruction ← getNextInstruction(vulnMethod)
end while
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.

Listing 4 shows another example of code which contains an early-exit timing side-channel vulnerability. The solution for early-exit timing side-channel vulnerabilities is to remove the early exit from the code. Listing 5 shows the corrected code (as

proposed by *DifFuzzAR*). The solution in this case consists in removing the early-exit, which was in line 17 in the original code, and changing the logic of the program to have a single exit point.

```
1  public static boolean passwordsEqual_unsafe(String a,
2  String b) {
3      boolean equal = true;
4      boolean temp = true;
5      boolean shmequal = true;
6      int aLen = a.length();
7      int bLen = b.length();
8      if (aLen != bLen) {
9          equal = false;
10     } else {
11         temp = false;
12     }
13     int min = Math.min(aLen, bLen);
14     for (int i = 0; i < min; i++) {
15         if (a.charAt(i) != b.charAt(i)) {
16            temp = true;
17            return false;
18         } else {
19             temp = false;
20            shmequal = true;
21         }
22     }
23     return equal;
24 }
```

Listing 4 Example of "Blazer PasswordEq" early-exit vulnerability before correction [7]

5.2 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.

Algorithm 4 shows 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.

Algorithm 4 Correction of Control-Flow Based Timing Side-Channel Vulnerabilities

```
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
    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()
```

```
public static boolean passwordsEqual_unsafe$Modification
                                    (String a, String b) {
    boolean $2 = true;
    boolean $1 = true;
    boolean equal = true;
    boolean shmequal = true;
    int aLen = a.length();
    int bLen = b.length();
    if (aLen != bLen) {
        equal = false;
    } else {
        $1 = false;
    }
    int min = Math.min(aLen, bLen);
    for (int i = 0; i < min; i++) {
        if (a.charAt(i) != b.charAt(i)) {
            $2 = true;
            equal = false;
        } else {
            $1 = false;
            shmequal = true;
        }
    }
    return equal;
}
```

Listing 5 Example of "Blazer PasswordEq" early-exit vulnerability after correction.

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

Listing 6 Example of method with control-flow based timing side-channel vulnerability [7]

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. An example is the code in Listing 6. Here, the method has a control-flow based timing side-channel vulnerability, since its execution time depends on the value of a secret, in this case the parameter `taint`. The 'then' and 'else' block of the 'if' statement has different instructions with different execution times. As such, to correct this vulnerability, the tool modifies the instructions of both blocks and adds the modified version of the 'then' block to the 'else' block and the modified version of the 'else' block to the 'then' block. The difference in the instructions is that the assignment is not made to the same variables, so as not to alter the value of the original variables. As a result, the tool produces the code in Listing 7.

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

Listing 7 Corrected version of method with control-flow based timing side-channel vulnerability

```
static boolean equals_unsafe(String expected, String actual
    ) {
  byte[] expectedBytes = bytesUtf8(expected);
  byte[] actualBytes = bytesUtf8(actual);
  int expectedLength = expectedBytes == null ? -1 :
      expectedBytes.length;
  int actualLength = actualBytes == null ? -1 : actualBytes
      .length;

  if (expectedLength != actualLength) {
    return false;
  }

  int result = 0;
  for (int i = 0; i < expectedLength; i++) {
    result |= expectedBytes[i] ^ actualBytes[i];
  }
  return result == 0;
}
```

Listing 8 Vulnerable method *equals_unsafe* (Themis Spring-Security)

5.3 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 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.

6 Evaluation

In this section, we describe how the developed tool was evaluated. The evaluation is divided in two main parts:

- testing the tool to ensure that the refactored code is semantically correct and to ensure that it has no timing side-channel vulnerabilities detected by *DifFuzz*, and
- a user study to assess if potential users would like to use *DifFuzzAR*, if they trust the code refactorings produced by the tool, and to gather suggestions for improvement.

6.1 Testing the tool

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 h. The results of the evaluation can be seen in Table 1.

6.1.1 Dataset used

We started with the 32 examples distributed with *DifFuzz*. One of the examples suffers from a size side-channel. For other examples, the correction seems to involve some domain-specific changes, which are outside the scope of our proposed method. For instance, the example *blazer modpow2*, seems to contain a control-flow based timing side-channel vulnerability that is corrected by changing the multiplication operation, which is outside the scope of our work. Since we are interested in assessing the effectiveness of our tool within the scope in

Table 1 Results of the application of *DifFuzzAR* to the *DifFuzz* dataset

Example name	Has secure version?	Type	Correction attempted	Semantically correct	Vulnerability corrected
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	Mixed	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

which it can operate, we decided to exclude these examples. 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.

One of the examples suffers from an early-exit timing side-channel vulnerability; eight of the examples contain a control-flow based timing side-channel vulnerability; the remaining 16 examples have a mixed timing side-channel vulnerability.

6.1.2 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 2020) 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 1 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. Regarding the missing variable declaration, one solution would be to add the creation of a new variable as the first instruction after the declaration of the variable that was replaced. However, this would add several extra instructions to the code and would create several instructions with the same name. All that would make it so that it would not be possible to minimize the execution time of the branches. It should be noted, however, that this can easily be fixed manually by the user.

6.1.3 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 1 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%.

Some of the attempted vulnerability corrections fail due to the unavailability of public arguments to fix control-flow vulnerabilities. For instance, consider the example *Themis Spring-Security* shown in Listing 8, which contains a mixed vulnerability: an early-exit vulnerability in line 8 and a control-flow vulnerability in line 12 (since the loop depends on the secret). For this example, our tool fixes the early-exit vulnerability, as shown in Listing 9. However, to repair the control-flow vulnerability, the tool attempts to replace the secret used in the stopping condition of the loop (*expectedLength*) with a public argument of the same type (as shown

in Algorithm 4). Since there are no public arguments of the same type, the tool does not change the stopping condition. This problem also applies to the examples *Themis Dynatable*, *Themis OrientDb*, *Themis PickedBox*. Without additional information, these examples cannot be fully repaired automatically. A possible future improvement is to bring the developer into the loop to obtain the additional information needed to repair the vulnerability. The other examples fail due to issues that go from the difficulty in identifying secrets due to complex data dependencies (e.g., in *Blazer Gpt14*, the secret is a parameter in the constructor of the class where the vulnerable method is defined) to the short circuiting semantics of conditional Java statements (which cause time variations when evaluating conditions, as in *Themis Boot-Stateless-Auth*). We have no immediate suggestions on how to fix these. More information about these failed corrections can be found at Lima's Master's thesis (Lima 2021).

```
1  static boolean equals_unsafe$Modification(String expected,
       String actual) {
2      int $3 = 0;
3      boolean $2;
4      boolean $1;
5      byte[] expectedBytes = bytesUtf8(expected);
6      byte[] actualBytes = bytesUtf8(actual);
7      int expectedLength = (expectedBytes == null) ? -1 :
           expectedBytes.length;
8      int actualLength = (actualBytes == null) ? -1 :
           actualBytes.length;
9      if (expectedLength != actualLength) {
10         $1 = false;
11     } else {
12         $2 = false;
13     }
14     int result = 0;
15     for (int i = 0; i < expectedLength; i++) {
16         if ((i < expectedLength) && (i < actualLength)) {
17             result |= expectedBytes[i] ^ actualBytes[i];
18         } else {
19             $3 |= 0;
20         }
21     }
22     $1 = result == 0;
23     return $1;
24 }
```

Listing 9 Attempted correction of *equals_unsafe* (Themis Spring-Security)

6.2 User study

In this section, we provide the motivation for the user study, its goals, and the methods used. We end the section with the results obtained.

6.2.1 Motivation

DifFuzzAR tries to repair timing side-channel vulnerabilities by automatically refactoring the Java code. To understand the impact that changes proposed by *DifFuzzAR* may have on users, we reviewed previous work on tools that automatically change code (i.e. refactoring tools).

Murphy-Hill et al. (2011) studied refactoring usage from Eclipse's user data. Their findings seem to suggest that users refrain from using refactoring tools because of three main factors:

1. lack of awareness of their existence;
2. lack of opportunity to use refactoring;
3. lack of trust in refactoring.

Users refrain from using refactoring tools in part because of a lack of trust about the full impact of the tools in the code. Murphy-Hill et al. (2011) stated that several developers mentioned they would avoid using a refactoring tool because of worries about introducing errors or unintended side-effects.

Another study by Eilertsen (2012) investigated the usability of refactoring tools. They interviewed and performed a usability study with 17 developers. They concluded that users of refactoring tools often complain about a lack of control and usability (Eilertsen 2012). Refactoring tools may change a program in unpredictable ways and users often like to review the code.

6.2.2 Goals

Given the human aspects affecting refactoring tools, we decided to perform a quantitative and subjective user study about *DifFuzzAR*. The goal is to understand if potential users would use our tool, if they trust its results, and what improvements they suggest. In particular, we are interested in answering the following research questions (RQs):

RQ1. Do users trust *DifFuzzAR*'s refactorings?

RQ2. What would users like to see in a tool such as *DifFuzzAR*?

RQ3. Do users value the use of *DifFuzzAR* differently in critical and non-critical applications?

6.2.3 Design and methods used

In order to answer our RQs we designed a survey study. We followed best practices from Redmiles et al. (2017) and followed methods similar to Eilertsen (2012). We recruited participants through our network (e.g., past students and colleagues).

The participants did not receive payment upon survey completion. Participants were shown a consent form before filling in the survey. They could remove their consent at any point without giving justification. We did not collect any personal data.

As we wanted to study potential users of our tool, we recruited participants with Java programming experience. We pre-screened participants and only accepted those that have been working with Java for at least two years in the previous 10 years. To characterize our sample, we also asked participants to rate their expertise in Java and if they knew what timing side-channel vulnerabilities were before our study.

We then asked all participants to go through a brief explanation of timing side-channel vulnerabilities with examples, and another explanation about *DifFuzzAR* and about what the tool does.

To understand if users would trust the results of our tool we provided four *vignette scenarios*. A vignette, as described in Lavrakas encyclopedia (Lavrakas 2008), describes a protagonist (or group of protagonists) faced with a realistic situation pertaining to the construct under consideration. The respondent is asked to make a judgment about the protagonist, the situation, or the correct course of action, using some form of closed-ended response. In our vignette scenarios, we presented the following scenario to participants:

> Imagine you apply DifFuzzAR to automatically repair timing side-channel vulnerabilities in Java code.

We then provided an example of code before and after a vulnerability is fixed by the tool. We used one scenario for control-based vulnerabilities, one for early-exit, and two for mixed vulnerabilities (one simple and one more complex). For each scenario, we asked participants to indicate, using a 5-point Likert agreement scale, if they would trust that refactoring. We also asked them for feedback. The code snippets used in these scenarios are taken from the examples used in the first part of the evaluation (Sect. 6.1) and can be seen in Appendix A. It is important to mention that in the early-exit example we modified the "before" code to remove the control-flow vulnerability. We did this to provide an example with only an early-exit vulnerability. While the "before" was adapted, the correction of the early-exit vulnerability showed to participants is a direct output of *DifFuzzAR*.

DifFuzzAR differs from usual refactoring tools as it aims at producing code that is more secure (instead of "cleaner" code). So, we also wanted to gauge if different situations impact users' willingness to use a tool like *DifFuzzAR*. With this goal in mind, we provided users with two scenarios of *DifFuzzAR*'s usage. One scenario described a programmer that is coding a sensitive part of a program (e.g., authentication) and the other describes a less critical situation (e.g., programming the GUI of an application). We call them critical and non-critical scenarios, respectively, and they can be reviewed in more detail in Appendix B. For each, we asked users to indicate their willingness to use *DifFuzzAR* using a 5-point Likert agreement scale. We also asked them for feedback. It should be noted that the non-critical scenario (focused on the dark mode change) can also leak information through observing timing differences, since the processing time for switching to dark mode might be slightly longer or shorter than switching to light mode. Despite the absence of secrets, in such a scenario, *DifFuzzAR* could potentially suggest a refactoring that

ensures a constant-time execution. Our goal in including this scenario is to understand whether developers see value in using *DifFuzzAR* in situations where no potentially sensitive information nor secrets can be retrieved. This can provide insights on how to best integrate tools such as *DifFuzzAR* into the developer's workflow.

We finished our survey by asking participants for suggestions to improve *DifFuzzAR* and by asking demographic questions.

User studies can suffer from bias due to the sample, how the survey questions are made, and even from the response options (Redmiles et al. 2017). To mitigate this problem, we did two cognitive interviews,[5] followed best practices by offering "don't know" or "prefer not to answer" responses (Redmiles et al. 2017), and used methods previously used in studies on related subjects (Eilertsen 2012).

6.2.4 Results

Our survey was answered by 20 users but only 11 meet our requirements and passed the pre-screening. To pass this pre-screening they needed to answer the question *"How many of the last ten years (2012-2022) have you spent developing or maintaining software in Java?"* with at least two years. We chose to do this because we wanted participants that could understand Java code well. Our sample of 11 participants has, on average, four years of experience with Java.

We followed the same methods as Eilertsen (2012) and asked participants to report their self-described proficiency with changing Java code. The respondents rated their proficiency on a scale from 1 to 5 (where 1 indicates no proficiency and five indicates expert proficiency). Most respondents (10 participants) reported at least three (i.e., above average) proficiency in Java. Our sample of users also reported a good familiarity with timing side-channel vulnerabilities as almost half of the respondents (45.45%) self-reported they knew what they were. Moreover, 27.27% of participants were not familiar with the concept and 27.27% were unsure. Most of the participants had a Master's degree (55%, i.e., 6 participants) with the remaining ones having a Bachelor's degree (45%). All participants were male with the exception of one female participant. The most common age group was 18-24 (63.6%), followed by 25-34 (18.2%), 35-44 (9.1%), and 45-54 (9.1%).

DifFuzzAR ***Scenarios***

Respondents went through four scenarios. For each scenario, they were presented with a concrete example of *DifFuzzAR*'s usage, where we provided Java code before and after applying the tool. After each scenario, we asked participants to use a 5-point Likert scale to indicate if they trust the refactorings produced by *DifFuzzAR*. When analyzing Likert scale data, it is recommended to use a non-parametric statistical test as the answers are not normally distributed (Lazar et al. 2017). Therefore, we analyzed the data using the Wilcoxon signed-rank non-parametric test with continuity correction. This test's null hypothesis is that there is not a significant difference between the samples. To disprove this hypothesis, the resulting *p-value* should

[5] Cognitive interviews involve asking respondents to think aloud as they complete a survey as well as asking them questions about each survey item (Redmiles et al. 2017).

be less than 0.05 ($p < 0.05$). When this happens, we conclude that there is a significant difference between the samples (Lazar et al. 2017).

The first two scenarios (control-flow and early-exit) were found to be very trustworthy by participants, i.e., most respondents stated they "agreed" or "strongly agreed" that they trusted the refactorings. Only one participant did not trust the early-exit scenario and, from their feedback, this was due to the wrong understanding that the refactoring did not preserve the behavior of the original code. Moreover, the Wilcoxon signed-rank non-parametric test did not find a statistical difference between the first two scenarios ($p > 0.05$).

The remaining two scenarios (mixed vulnerabilities, one simple and one more complex) while generally trusted (55% "agreed" or "strongly agreed" that they trusted the refactorings) had a more mixed response from participants. Two (2) out of 11 participants stated they did not trust the refactoring done in scenarios 3 and 4. Their answers seem to indicate that this may be due to two reasons: (a) the participant was unable to understand the changes that had been made to the code (e.g., the participant did not understand the final version of the code); (b) the scenario was complex (e.g., the participant associated complexity with distrust). This lack of trust in more complex refactorings is something that previous literature on refactoring also found to be the case (Eilertsen 2012).

While these scenarios (3 and 4) were not found as trustworthy as the first ones (1 and 2) there was no statistical difference between any of them ($p > 0.05$).

Critical and non-critical scenarios

Now that we have established that most participants trust *DifFuzzAR*'s refactorings, we turn our attention to the critical and non-critical scenarios. The critical scenario describes programming authentication code and the non-critical describes programming an application's GUI. All users found *DifFuzzAR* to be useful in the critical scenario (100% answered "agree" or "strongly agree") but to be less useful in the non-critical scenario (27% "agree" or "strongly agree"). These results are statistical different with $p = 0.00072$.

After analyzing participants' answers, our results suggest that users value more *DifFuzzAR*'s usage in critical use cases, and less in day-to-day coding procedures as one participant stated "In the second (scenario), there is no risk associated with the functionality, no secrets are involved.". The results also seem to indicate that the prevention of timing side-channel vulnerabilities is more important when their exploitation can lead to loss of valuable secrets (like in the critical scenario).

Further improvements

After the previous questions, we asked participants to suggest improvements and future features that *DifFuzzAR* could implement. We coded their answers with an *emergent coding scheme* (Lazar et al. 2017) as we had no previous insights about what their answers could be. The frequency of their answers and codes can be seen in Table 2. It is important to note that respondents' answers may be coded with more than one code and some users may have not answered at all, so, the total frequency of the codes does not necessarily match the number of participants in the study.

Some participants stated that they would like to see additional features in *DifFuzzAR* such as the correction of other common vulnerabilities in other languages.

Table 2 Coded answers to "What functionalities/ improvements would you like to see in this tool?" and frequency of answers

Code description	Frequency
Correct other common vulnerabilities	3
Provide more information about the changes	3
I don't know	2
Add analysis of nested functions	1
Less complex changes	1
Apply the tool to interpreted languages (e.g., Python)	1
Improve variable names	1

One of the participants even goes as far as suggesting they would like "to do this (use *DifFuzzAR*) in an interpreted language (e.g., Python) instead of a compiled language". Another common theme in users' suggestions for improvements was that *DifFuzzAR* could provide more information about the changes it makes (e.g., by adding comments explaining the changes, with better variable names or, as one user mentioned, with a "detailed profiling of the modified code, before and after it was modified (...)".) Overall users' comments were positive and their feedback suggested useful future features for *DifFuzzAR*.

6.2.5 Discussion

We were able to successfully answer the proposed research questions (see Sect. 6.2.2) and the insights gathered seem to confirm the usefulness of *DifFuzzAR*. We now address briefly each of our research questions.

RQ1. Do users trust DifFuzzAR's *refactoring?*

Our results suggest that users trust *DifFuzzAR*'s code transformations. They also seem to indicate that the complexity of the transformations affect trust, as simpler code transformations were seen as more trustworthy by participants.

RQ2. What would users like to see in a tool such as DifFuzzAR? This study's results seem to indicate that our participants value

a thorough tool that also corrects other common vulnerabilities in other programming languages. Users' also seem to value a tool that is transparent about its functioning, this is, a tool that informs the user about the changes it does to the code.

RQ3. Do users value the use of DifFuzzAR differently in critical and non-critical applications?

Our results seem to suggest that users value the use of *DifFuzzAR* differently in distinct situations. From the data gathered, participants appear to be more willing to use *DifFuzzAR* in more critical Java code. If this is the case, then the potential impact of *DifFuzzAR* is greater as this type of code is impactful to the security of a product.

While we gathered significant data in this user study, this data is subjective as our sample size is relatively small. However, this study has still provided valuable insights that can inform future large-scale user studies. It also seems to confirm that users generally trust refactorings produced by *DifFuzzAR* and that they see value in

a tool like this, in particular in more critical Java code. It has also provided suggestions for improvement of *DifFuzzAR* that can also be useful for other similar tools.

7 Threats to validity

Regarding our tool, a threat to validity is that it can have bugs and be incapable of fixing vulnerable code not considered in this study. We mitigate this risk by explicitly stating the scope of our tool (i.e., the goal is to work in conjunction with *DifFuzz*) and by considering in our study all the examples available in *DifFuzz*'s public benchmark. Moreover, all our code and data are publicly available for other researchers and potential users to check the validity of the results.

Regarding the user study, a threat to validity is the fact that we recruited participants through our professional network (e.g., past students and colleagues), as this can potentially bias the results. However, the sampling procedure that we use is commonly employed in studies of this nature. For example, as one of the reviewers of this work brought to our attention, works that can be used as guidelines for empirical studies, such as Ciolkowski et al. (2003), use similar approaches as they use their own industrial contacts.

While the user study might have limitations and the results might not be completely generalizable, it nonetheless offers valuable insights to the community and has the potential to shape and guide future research endeavors. We do not claim that our study is perfect, but we argue that it provides valuable information. Moreover, we followed best practices from Redmiles et al. (2017) and followed methods similar to Eilertsen (2012). Similar methods are also used in other research papers. For example, we use vignette-based surveys, as such surveys have been found to well-approximate real-world behavior (Cummings et al. 2021; Hainmueller et al. 2015). It is worth noting that, as stated by Berry and Tichy (2003), a reference kindly suggested by one of the reviewers of this paper, *"Perfection in experiments, especially in those involving human subjects, is unattainable"*.

8 Conclusions

This paper presents a tool for automatic repair of timing side-channel vulnerabilities in Java code that works in conjunction with *DifFuzz* (Nilizadeh et al. 2019). 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* (Nilizadeh et al. 2019), 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. Moreover, the results obtained in a quantitative and subjective user study seem to confirm that users generally trust refactorings produced by *DifFuzzAR* and that they see value in a tool like this, in particular in more critical Java code.

The fact that users do not seem to value the use of *DifFuzzAR* in non-critical scenarios suggests that one should be careful when integrating a tool of this nature into the developer's workflow. It might not be advisable to allow such a tool to provide refactoring suggestions for *every* component that might leak information through observing timing differences, as fixing non-critical issues might frustrate developers and reduce the adoption of such a tool. A more judicious use, where only components that deal with secrets are considered, is recommended. This might be achieved by attempting to automatically infer the secrets or by allowing the developers to manually annotate the code to identify such secrets.

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.

8.1 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.

8.2 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 Sect. 4. 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 submit any corrections found as pull requests, thus improving existing software and, simultaneously, obtaining code reviews from developers (as done in related refactoring projects by the authors Ribeiro et al. 2021; Pereira et al. 2022).

Further user studies can also be useful for future improvements of *DifFuzzAR*. The user study presented here allowed us to gather significant data, but this data is subjective due to the size of the sample. As such, we suggest that future work should repeat our study with a larger sample of users. Future work on this topic should also include a usability analysis of *DifFuzzAR*'s usage with a larger number of users. We also believe that more work should be done to understand users' motivations when using software to automatically repair vulnerabilities. While there have been previous works done about refactoring there is still a gap in the literature when it comes to tools os the same nature as *DifFuzzAR*.

Appendix 1: Java Code snippets used in user study scenarios

```
1  public static boolean array_unsafe(int a[], int taint) {
2      System.out.println(a.length);
3      int t;
4      if (taint < 0) {
5          int i = a.length-1;
6          // int i = a.length;
7          while (i >= 0) {
8              t = a[i];
9              i--;
10         }
11     } else {
12         int i = 0;
13         t = a[i] / 2;
14         i = a.length;
15     }
16     return false;
17 }
```

Listing 10 Scenario 1. Example of "Blazer Array" control-flow vulnerability before correction.

```
1  public static boolean array_unsafe$Modification(int[] a,
2  int taint) {
3      int $2;
4      System.out.println(a.length);
5      int t;
6      if (taint < 0) {
7          int $3 = 0;
8          $2 = a[$3] / 2;
9          $3 = a.length;
10         int i = a.length - 1;
11         while (i >= 0) {
12             t = a[i];
13             i--;
14         }
15     } else {
16         int $1 = a.length - 1;
17         while ($1 >= 0) {
18             $2 = a[$1];
19             $1--;
20         }
21         int i = 0;
22         t = a[i] / 2;
23         i = a.length;
24     }
25     return false;
26 }
```

Listing 11 Scenario 1. Example of "Blazer Array" control-flow vulnerability after correction.

```
1  public static boolean passwordsEqual_unsafe(String a,
2  String b) {
3      boolean equal = true;
4      boolean temp = true;
5      boolean shmequal = true;
6      int aLen = a.length();
7      int bLen = b.length();
8      if (aLen != bLen) {
9          equal = false;
10     } else {
11         temp = false;
12     }
13     int min = Math.min(aLen, bLen);
14     for (int i = 0; i < min; i++) {
15         if (a.charAt(i) != b.charAt(i)) {
16            temp = true;
17            return false;
18         } else {
19             temp = false;
20            shmequal = true;
21         }
22     }
23     return equal;
24 }
```

Listing 12 Scenario 2. Example of "Blazer PasswordEq" early-exit vulnerability before correction.

```
public static boolean passwordsEqual_unsafe$Modification
(String a, String b) {
    boolean temp = true;
    boolean equal = true;
    boolean shmequal = true;
    int aLen = a.length();
    int bLen = b.length();
    if (aLen != bLen) {
        equal = false;
    } else {
        temp = false;
    }
    int min = Math.min(aLen, bLen);
    for (int i = 0; i < min; i++) {
        if (a.charAt(i) != b.charAt(i)) {
            temp = true;
            equal = false;
        } else {
            temp = false;
            shmequal = true;
        }
    }
    return equal;
}
```

Listing 13 Scenario 2. Example of "Blazer PasswordEq" early-exit vulnerability after correction.

```
public boolean equals(Object other) {
    if (this == other) {
        return true;
    }
    if (other == null || getClass() != other.getClass())
    {
        return false;
    }

    Impl impl = (Impl) other;

    return ArraysIsEquals(password, impl.password);
}
```

Listing 14 Scenario 3. Example of "OACC" mixed timing vulnerability before correction.

```
public boolean equals$Modification(Object other) {
    boolean $2;
    boolean $1;
    if (this == other) {
        $1 = true;
    } else {
        $2 = true;
    }
    if ((other == null) || (getClass() != other.getClass()))
    {
        $1 = false;
    } else {
        $2 = false;
    }
    Impl impl = ((Impl) (other));
    $1 = ArraysIsEquals(password, impl.password);
    return $1;
}
```

Listing 15 Scenario 3. Example of "OACC" mixed timing vulnerability after correction.

```
public boolean isEqual_unsafe(String thisObject,
Object otherObject) {
        if (thisObject == otherObject) {
            return true;
        }
        if (otherObject instanceof String) {
            String anotherString = ((String) (otherObject));
            int n = thisObject.length();
            if (n == anotherString.length()) {
                char[] v1 = thisObject.toCharArray();
                char[] v2 = anotherString.toCharArray();
                int i = 0;
                while ((n--) != 0) {
                    if (v1[i] != v2[i])
                        return false;

                    i++;
                }
                return true;
            }
        }
        return false;
    }
```

Listing 16 Scenario 4. Example of "AuthmReloaded" mixed timing vulnerability before correction.

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

Listing 17 Scenario 4. Example of "AuthmReloaded" mixed timing vulnerability after correction.

Appendix 2: Scenarios in user study

In the user study, participants were presented with two scenarios, one critical and one non-critical. These scenarios are presented below.

Critical Scenario

Imagine you are developing an application in Java. You write the code that is used for the login module of your project. Your code receives a user-provided string and compares it with a secret password. After writing the code, you think about whether you should use DifFuzzAR.

Non-critical Scenario

Imagine you are developing an application in Java. You write the code that is used for the GUI module of your project. Your code receives user-input when a user clicks on the "dark mode" button and changes the interface accordingly. After writing the code, you think about whether you should use DifFuzzAR.

Acknowledgements We thank the anonymous reviewers of the previous version of this work (Lima et al. 2021) for their valuable and constructive comments. We also thank the anonymous reviewers of this extended version for their comments, which substantially improved the paper. 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.

Funding Open access funding provided by FCT|FCCN (b-on).

References

Allamanis, M., Jackson-Flux, H., Brockschmidt, M.: Self-supervised bug detection and repair. In: NeurIPS (2021)

Antonopoulos, T., Gazzillo, P., Hicks, M., Koskinen, E., Terauchi, T., Wei, S.: Decomposition instead of self-composition for proving the absence of timing channels. ACM SIGPLAN Notices **52**(6), 362–375 (2017)

Berry, D.M., Tichy, W.F.: Comments on "Formal methods application: an empirical tale of software development''. IEEE Trans. Softw. Eng. **29**(6), 567–571 (2003)

Brumley, D., Boneh, D.: Remote timing attacks are practical. Comput. Netw. **48**(5), 701–716 (2005)

Chen, J., Feng, Y., Dillig, I.: Precise detection of side-channel vulnerabilities using quantitative Cartesian Hoare logic. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 875–890. ACM (2017)

Chen, Z., Kommrusch, S.J., Tufano, M., Pouchet, L.-N., Poshyvanyk, D., Monperrus, M.: Sequencer: sequence-to-sequence learning for end-to-end program repair. IEEE Trans. Softw. Eng. **47**, 1943–1959 (2019)

Chen, Z., Kommrusch, S., Monperrus, M.: Neural transfer learning for repairing security vulnerabilities in C code. arXiv preprint arXiv:2104.08308 (2021)
Ciolkowski, M., Laitenberger, O., Vegas, S., Biffl, S.: Practical Experiences in the Design and Conduct of Surveys in Empirical Software Engineering. Springer, Berlin (2003)
Cloud Foundry: These are the top languages for enterprise application development and what that means for business. Accessed 2020-08-17. https://www.cloudfoundry.org/wp-content/uploads/Developer-Language-Report_FINAL.pdf
Cornu, B., Durieux, T., Seinturier, L., Monperrus, M.: Npefix: Automatic runtime repair of null pointer exceptions in java. arXiv preprint arXiv:1512.07423 (2015)
Cummings, R., Kaptchuk, G., Redmiles, E.M.: "I need a better description": an investigation into user expectations for differential privacy. In: Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security, pp. 3037–3052 (2021)
DARPA: Space/time analysis for cybersecurity (STAC). Accessed 2020-08-17. https://www.darpa.mil/program/space-time-analysis-for-cybersecurity
Eilertsen, M.: Improving the usability of refactoring tools for software change tasks. Ph.D. thesis, University of Bergen (2012)
EvoSuite: Automatic test suite generation for Java. Accessed 2020-08-27. https://www.evosuite.org/
Forrest, S., Nguyen, T., Weimer, W., Le Goues, C.: A genetic programming approach to automated software repair. In: Proceedings of the 11th Annual Conference on Genetic and Evolutionary Computation, pp. 947–954 (2009)
GitHub: the state of the octoverse. Accessed 2019-10-07. https://octoverse.github.com/projects#languages
Goguen, J.A., Meseguer, J.: Security policies and security models. In: 1982 IEEE Symposium on Security and Privacy, p. 11. IEEE (1982)
Goues, C.L., Pradel, M., Roychoudhury, A.: Automated program repair. Commun. ACM **62**(12), 56–65 (2019)
Gupta, R., Pal, S., Kanade, A., Shevade, S.: Deepfix: fixing common C language errors by deep learning. In: Thirty-First AAAI Conference on Artificial Intelligence (2017)
Hainmueller, J., Hangartner, D., Yamamoto, T.: Validating vignette and conjoint survey experiments against real-world behavior. Proc. Natl. Acad. Sci. **112**(8), 2395–2400 (2015)
IBM: Modern languages for the modern enterprise. Accessed 2020-08-17. https://developer.ibm.com/articles/d-modern-language-modern-enterprise/
IVC Wiki: Xbox 360 timing attack. Accessed 2020-08-17. https://beta.ivc.no/wiki/index.php/Xbox_360_Timing_Attack
Kersten, R., Luckow, K., Păsăreanu, C.S.: Poster: Afl-based fuzzing for java with kelinci. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, pp. 2511–2513. ACM (2017)
Kim, D., Nam, J., Song, J., Kim, S.: Automatic patch generation learned from human-written patches. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 802–811. IEEE (2013)
Koeune, F., Standaert, F.-X.: A tutorial on physical security and side-channel attacks. In: Foundations of Security Analysis and Design III, pp. 78–108. Springer, Berlin (2005)
Lavrakas, P.J.: Encyclopedia of Survey Research Methods. Sage publications, Los Angeles (2008)
Lawson, N.: Timing attack in Google Keyczar library. Accessed 2020-08-17. https://rdist.root.org/2009/05/28/timing-attack-in-google-keyczar-library/
Lazar, J., Feng, J.H., Hochheiser, H.: Research Methods in Human–computer Interaction. Morgan Kaufmann, Boston (2017)
Lima, R., Ferreira, J.F., Mendes, A.: Automatic repair of Java code with timing side-channel vulnerabilities. In: 2021 36th IEEE/ACM International Conference on Automated Software Engineering Workshops (ASEW), pp. 1–8 (2021). https://doi.org/10.1109/ASEW52652.2021.00014
Le Goues, C., Nguyen, T., Forrest, S., Weimer, W.: Genprog: a generic method for automatic software repair. IEEE Trans. Softw. Eng. **38**(1), 54–72 (2011)
Lima, R.: Automatic repair of Java code with timing side-channel vulnerabilities. Master's thesis, Instituto Superior Técnico, University of Lisbon (January 2021). https://fenix.tecnico.ulisboa.pt/cursos/meic-t/dissertacao/1128253548921982
Liu, K., Koyuncu, A., Kim, D., Bissyandé, T.F.: Tbar: revisiting template-based automated program repair. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 31–42 (2019)

Lutellier, T., Pham, H.V., Pang, L., Li, Y., Wei, M., Tan, L.: Coconut: combining context-aware neural translation models using ensemble for program repair. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 101–114 (2020)

Mechtaev, S., Yi, J., Roychoudhury, A.: Angelix: scalable multiline program patch synthesis via symbolic analysis. In: Proceedings of the 38th International Conference on Software Engineering, pp. 691–701 (2016)

Monperrus, M.: Automatic software repair: a bibliography. ACM Comput. Surv. (2015). https://doi.org/10.1145/3105906

Monperrus, M.: The living review on automated program repair. Technical Report hal-01956501, HAL Archives Ouvertes (2018). https://www.monperrus.net/martin/repair-living-review.pdf

Murphy-Hill, E., Parnin, C., Black, A.P.: How we refactor, and how we know it. IEEE Trans. Softw. Eng. **38**(1), 5–18 (2011)

Nguyen, H.D.T., Qi, D., Roychoudhury, A., Chandra, S.: Semfix: program repair via semantic analysis. In: 2013 35th International Conference on Software Engineering (ICSE), pp. 772–781. IEEE (2013)

Nilizadeh, S., Noller, Y., Păsăreanu, C.S.: Diffuzz: differential fuzzing for side-channel analysis. In: Proceedings of the 41st International Conference on Software Engineering, pp. 176–187. IEEE Press (2019)

Pawlak, R., Monperrus, M., Petitprez, N., Noguera, C., Seinturier, L.: Spoon: a library for implementing analyses and transformations of Java source code. Softw. Pract. Exp. **46**, 1155–1179 (2015). https://doi.org/10.1002/spe.2346

Pereira, R.B., Ferreira, J.F., Mendes, A., Abreu, R.: Extending EcoAndroid with automated detection of resource leaks. In: 9th IEEE/ACM International Conference on Mobile Software Engineering and Systems 2022 (MobileSoft) (2022)

Redmiles, E.M., Acar, Y., Fahl, S., Mazurek, M.L.: A summary of survey methodology best practices for security and privacy researchers. Technical report (2017)

Ribeiro, A., Ferreira, J.F., Mendes, A.: EcoAndroid: an android studio plugin for developing energy-efficient Java mobile applications. In: 2021 IEEE 21st International Conference on Software Quality, Reliability and Security (QRS), pp. 62–69 (2021). https://doi.org/10.1109/QRS54544.2021.00017

Wu, M., Guo, S., Schaumont, P., Wang, C.: Eliminating timing side-channel leaks using program repair. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, pp. 15–26 (2018)

Xuan, J., Martinez, M., Demarco, F., Clement, M., Marcote, S.L., Durieux, T., Le Berre, D., Monperrus, M.: Nopol: automatic repair of conditional statement bugs in java programs. IEEE Trans. Softw. Eng. **43**(1), 34–55 (2016)

Yasunaga, M., Liang, P.: Graph-based, self-supervised program repair from diagnostic feedback. In: International Conference on Machine Learning, pp. 10799–10808. PMLR (2020)

Yasunaga, M., Liang, P.: Break-it-fix-it: unsupervised learning for program repair. In: International Conference on Machine Learning (ICML) (2021)

Ye, H., Martinez, M., Monperrus, M.: Neural program repair with execution-based backpropagation. arXiv preprint arXiv:2105.04123 (2021)

Zalewski, M.: American fuzzy lop (2017)

Zhou, Y., Feng, D.: Side-channel attacks: ten years after its publication and the impacts on cryptographic module security testing. IACR Cryptol. ePrint Archive **2005**, 388 (2005)