5-2018

Tracing Vulnerabilities Across Product Releases

Adriana Sejfia
axs1461@rit.edu

Tracing Vulnerabilities Across Product Releases

by

Adriana Sejfia

THESIS

Presented to the Faculty of the Department of Software Engineering

Golisano College of Computer and Information Sciences

Rochester Institute of Technology

in Partial Fulfillment

of the Requirements

for the Degree of

Master of Science in Software Engineering

Rochester Institute of Technology

May 2018

Tracing Vulnerabilities Across Product Releases

APPROVED BY

SUPERVISING COMMITTEE:

Dr. Mehdi Mirakhorli, Supervisor

Dr. Christian D. Newman, Reader

Dr. J. Scott Hawker, Graduate Program Director

Dedicated to my Mom. This and everything else.

Acknowledgments

I wish to thank the multitudes of people who helped me in this journey.

I would like to thank my adviser, Dr. Mehdi Mirakhorli, for his timely and thorough advice. I would also like to thank Dr. Christian Newman for accepting to be in my thesis committee and for his valuable feedback.

I would like to thank the reviewers, who tirelessly looked over all the data and ensured a smooth evaluation process.

I would like to thank the faculty and staff at Golisano for having made this journey as intriguing as possible.

I would like to thank my family, friends and loved ones for the unconditional support.

Abstract

Tracing Vulnerabilities Across Product Releases

Adriana Sejfia, M.S.
Rochester Institute of Technology, 2018

Supervisor: Dr. Mehdi Mirakhorli

When a software development team becomes aware of a vulnerability, it generally only knows that the last version of that software product is vulnerable. However, today most software products have more than one version being actively used at a time. Garnering information on which versions contain a vulnerability, and which do not, is crucial for the users, to know which versions of a software product are safe to use, and also for the developers, to know where to apply the patch. The patch, i.e. the fix of the vulnerability, contains valuable information in the form of changes made to the known vulnerable code to fix it. This information could be leveraged to analyze the presence of this known vulnerability across releases of a software product. The problem of tracing vulnerabilities in different releases has been addressed in two separate research projects. Both of these projects rely on the changed

lines of code to fix a vulnerability, and conclude whether a version is vulnerable or not based on the presence of these lines of code. However, relying simply on lines of code fails to consider the changes in the source code context where the patch has been introduced from a version to a version. In addressing this problem, this research project will focus on representing the patch and the versions to be evaluated in a more flexible format such as an Abstract Syntax Tree (AST). This approach is more robust compared to the line-based approach, because ASTs abstract away these changes in the context and allow us to focus more efficiently on the structure and behavior of the code in the patch. As such, instead of using lines of code, the unit of comparison in our approach will be nodes in an AST. Moreover, our approach will generate comprehensive artifacts that could guide developers to more efficiently patch the different versions of their product. We implemented our approach in a Java tool named Patchilyzer and we tested it in 174 Tomcat versions for a total of 39 vulnerabilities.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

The multitude of software products has swept the financial, health care, and even shopping domains. Customers in these domains become users of software products, which often time translates to them providing their personal information, e.g. credit cards, social security numbers, and addresses, to these programs. Moreover, the smooth running of these software solutions is required to successfully carry out critical processes, varying from stock exchange to surgeries. With the ever-increasing permeation of software in everyday processes, ensuring its security becomes a highly important task. Lack of security in a widely used software could mean that critical processes are interrupted or that personal information of users is leaked to malicious parties.

In January, 2017, 1141 vulnerabilities[1] have been reported in the National Vulnerability Database(NVD). In January, 2018, 1716[2] vulnerabilities were reported in the same database. The same increasing trend is noted in February and March as well[3]. The increasing trend of vulnerabilities is concerning. Parallel to that, the pressure that software developers face to mini-

[1]https://nvd.nist.gov/vuln/full-listing/2017/1
[2]https://nvd.nist.gov/vuln/full-listing/2018/1
[3]https://nvd.nist.gov/vuln/full-listing

mize the risk for vulnerabilities is increasing as well. As such, there is a need to speed up the process of fixing vulnerabilities and helping developers to deal with the ever-increasing burden of ensuring the security of their products. In other words, there is a need for automated solutions that help in detecting vulnerabilities, among other issues.

Once a vulnerability is identified in the source code of a software product, the developers usually tend to it by creating and testing a patch that fixes the vulnerability. Nonetheless, usually, this patch or fix is released only for the particular version of that software product in which that vulnerability was found. This is so despite it being a widely employed practice in the industry to have more than one version of a software product available for users at a time. On the one hand, since versions of the software product usually share at least some parts of source code, it could happen that other versions are vulnerable to the same vulnerability. At the same time, users could be using versions other than the one in which the vulnerability has been found and fixed. In fact, a study found out that the most of vulnerabilities in Firefox stem from unmaintained code in older versions [9]. While these versions are not patched, they could be exploited by malicious parties. Moreover, there is always the possibility that, in newer versions, with rolling changes, the source code can regress and the vulnerability could be re-introduced.

On the other hand, it could very well be that the vulnerability is not present in the earlier versions. NVD, one of the most reputable online sources that tags software products as vulnerable or not has been shown to make

'spurious claims' regarding the vulnerable status of a software product [13, 14]. These spurious claims could make users of these products stop using them or go through the expensive process of switching to other products, unnecessarily.

Considering this information, it is clear that there is a need to check for the presence of a vulnerability in versions other than the one in which that vulnerability was found. Performing this task manually would be tedious and time-consuming for the developers. Hence, one approach to solve this problem would be to automate the process of checking for a given vulnerability in different versions of the software product.

There has been studies that attempted to solve this problem. These previous research projects focused on leveraging information from the patch in terms of lines added and/or deleted in the source code to fix the vulnerability [1, 13]. This approach works as follows: if added lines in the patch exist in another version, that is proof that the vulnerability might not exist in that version; if deleted lines in the patch exist in another version, that is proof that the vulnerability might exist in that version. Although, this approach laid the groundwork in this particular area, it doesn't consider the flexibility of the source code from one version to the other. While source code is shared between versions, there are still changes between versions that are not relevant to the vulnerability. Simply looking at the lines of code would not take into account ways in which the vulnerability or the fix in another version could change to fit the context in that particular version. To address this problem, while automating the check for the presence of a given vulnerability, we pro-

pose leveraging information from the patch in an Abstract Syntax Tree (AST) representation that would allow for more comprehensive, flexible and accurate checks of the presence of vulnerable nodes, as compared to vulnerable lines. Moreover, our approach would produce comprehensive artifacts for developers to analyze that would help in pinpointing the presence of a vulnerability, such as highlighting the presence of vulnerable nodes, or nodes similar to those, in different releases.

Chapter 2

Background

2.1 Vulnerability Detection and Prediction

A software development team in charge of a product can come to know of vulnerabilities in different ways. Users, third parties, or the team itself can come across an attack vector that could be used to exploit the code. Once the team obtains that knowledge, generally, it starts working through first, identifying the cause of the problem and second, finding efficient and appropriate solutions to the problem. Varying teams have different tools at their disposal to help them in this process. Once a solution is created for the problem, it is implemented in terms of changes such as deletions, additions and modifications of the existing source code. The set of all changes made to fix a vulnerability is known as the patch. As seen in Figure 2.1, part of a patch to fix one of the vulnerabilities in Tomcat, a web server, from its github mirror repository, consists of adding and deleting several lines of code. This is just one part of the patch, as the full patch contains modifications to more than one file.

Even though patching a software vulnerability is of utmost importance, studies have shown that there are several factors, such as legal factors, eco-

nomical factors and the type of vulnerability, that contribute to when patches get released and what gets patched [18]. In the same paper, it is mentioned that when a vulnerability was found in Tomcat, the first version to be fixed was the last one, whereas two other older versions that were also affected by the vulnerability were fixed one, respectively seven months later. One way developers could be incentivized to patch all the other versions as soon as possible, besides the economical factors, would be to facilitate the process of knowing these versions and provide them with automated approaches that could highlight vulnerable code elements for them. This is one of the goals of this project.

One of the reasons behind the differences in the timing of releases of patches is that the security issues developers face with are so numerous and complex that they might end up taking a lot of resources. The resources the teams have are finite and researchers have taken it upon themselves to try to help developers to make better use of them. For instance, Theisen et al. [19] try to develop a technique that would enable software developers to reduce the risk surface attack of their products. This means that developers would try early on to reduce the entry points to their program that could be exploited by vulnerabilities. Researchers have also tried to predict when a vulnerability will be discovered to help the developers plan their resources ahead to tackle these issues [8, 21]. These papers overlap with our work in so far that the purpose of these projects is to reduce the workload of developers; however, our approach tries to do so by helping developers tag versions as vulnerable or not through

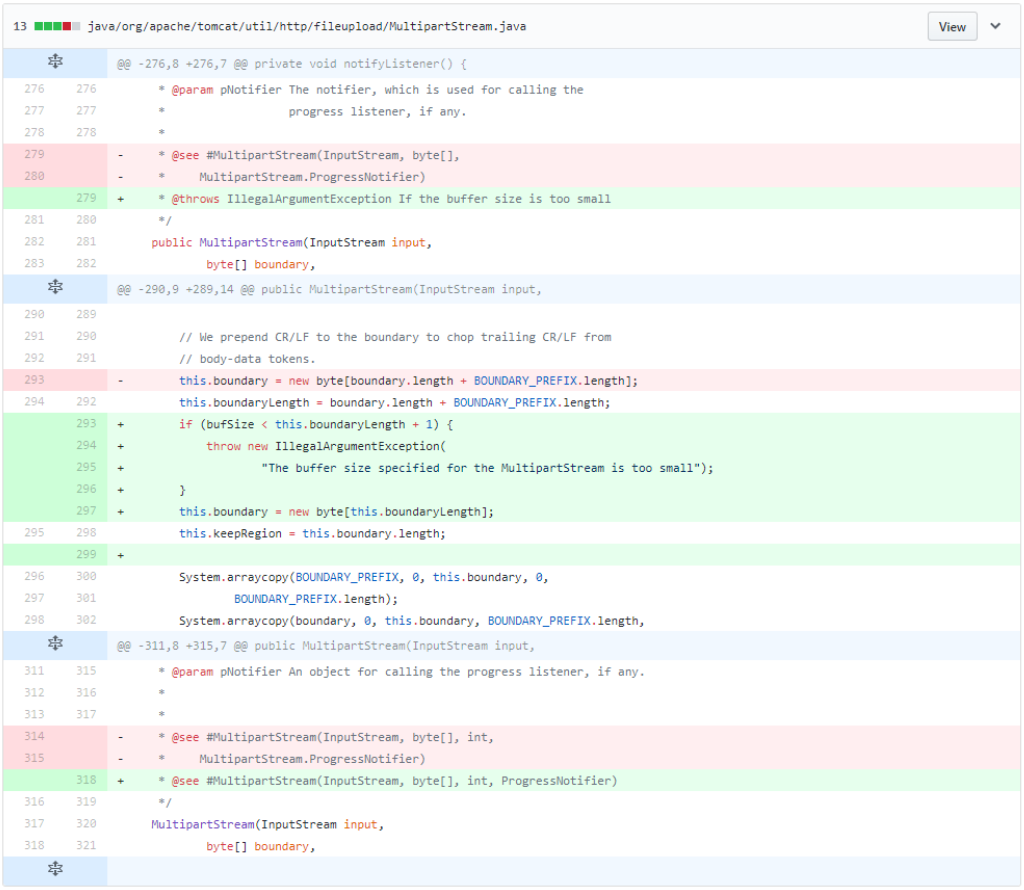

Figure 2.1: Partial Patch for a CVE in Tomcat

an automated approach that would replace intensive manual effort.

The issue of tracing known vulnerabilities across releases of a software product has been tackled in previous work as well. Specifically, other projects have also been focused in leveraging information from the patch. In his work, Craig [1] sets out a technique to trace vulnerabilities leveraging information on the patch. The technique divides the patch in two parts: one part considers only the additions and the other only the deletions. Having these two artifacts,

the codebase of the subject version is evaluated twice: once for the presence of the elements in the additions part of the patch, and once for presence of the elements in the deletions part. After obtaining information on what lines are present and what lines are not in the codebase, the technique uses thresholds set by the author to conclude if a vulnerability is present or not. Similarly, the method developed by Nguyen et al. [13] traces vulnerabilities across releases by looking at the vulnerable code footprint, the equivalent of patch. Again, the method looks for the presence of the vulnerability footprint in versions. Looking retroactively at older versions, it tries to see at which version exactly did the vulnerability start to be introduced in the program. All versions after that point are considered vulnerable, and all before that are considered as clean-slate. Both these approaches rely on lines of code as their method of comparison. Our approach, on the other hand, lies on AST representation of the source code. Moreover, on top of the similarities' check we also look at whether the nodes that contained the vulnerability in the first place were there. This additional check is to make sure that we do not tag versions that did not contain the vulnerability in the first place as vulnerable.

Previously, there has been work in helping developers detect any vulnerability, such as constraint-solving methods [20]. This method relies on a search-driven technique to solve constraints for string variables in order to detect potential vulnerabilities related to these variables. Moreover, there have been studies that focus on detecting code clones as a means to detect known buggy code across android applications [5]. In another work, the focus was do-

main specific, namely looking at code clones in Operating Systems [7]. Other studies have focused in predicting vulnerabilities using text analysis on source code [6], using code metrics such as code churn and complexity as indicators for vulnerabilities [2, 17], using the past history of bugs [10], or tagging components that are more likely to be vulnerable [11]. Even though there is an overlap between our project and these projects, the goal of our work is different. Our approach leverages information about a known vulnerability and attempts to trace that vulnerability in other versions, in a more flexible manner than detecting code clones. Our method also is not limited in terms of the existing data types it uses.

2.2 Alternative Code Representations

Previous studies have used AST representations of the source code to increase the flexibility of their approaches, especially when considering changes from one version to another version. For instance, Nguyen et al. used AST representations to create statistical models for changes that co-occur in code together to be able to give automatic API recommendations [13]. The changes specifically are represented in AST format. While this paper focuses on API recommendations, our approach focuses on representing those changes in the most flexible manner, while not losing on accuracy, to enable a more thorough comparison of the said changes to tag vulnerable versions. Moreover, Zhang and Liu in their work use AST representations to detect code plagiarism [16]. While this paper bears some similarities to our approach, our goal is not

simply to detect if code in two versions is the same, but rather to also detect which parts are similar, and which are not, and how do these two pieces of information enable us to conclude if a version is vulnerable or not.

In terms of comparing trees for similarities, previous studies have considered using edit distances. Pawlik and Augsten use the All Path Tree Edit Distance to calculate differences between trees [15]. Moreover, Fischer et al. [4] in their work use Hausdorff matching to approximate graph edit distances. However, these techniques are not suitable for the purpose of our approach. The changes that we will see through the AST differencing output will be in terms of added, deleted, moved or updated node. To compare the differences from a version to another version, the node types and actions, at the least, need to perfectly match from a version to another. That is why the similarity check relies on a perfect similarity in terms of individual nodes, when it comes to types and the action that was performed on those nodes.

The technique developed by Falleri et al. [3] GumTree, is used to produce a fine-grained differencing output between two source code artifacts in an AST format using nodes and edges. The technique developed by this paper works in generating the fine-grained differencing output that we need for our approach.

Chapter 3

Methodology

3.1 Definitions

In order to facilitate the understanding of the approach, the following terms need to be defined:

1. **Fixed Version:** A fixed version F is the version in which the vulnerability was fixed. In order to eliminate noise-introducing changes that are not related to the fix of the vulnerability, the term version refers to the exact revision in which the vulnerability was fixed.

2. **Vulnerable Version:** A vulnerable version V is the version in which the vulnerability has been found. The term version here refers to the exact revision of the software product in which the vulnerability was found, aligning with our goal of eliminating changes not related to the vulnerability from the picture. The vulnerable revision is identified as the revision previous to the fixed version chronologically, unless stated otherwise.

3. **Subject Version:** A subject version X is the version to be evaluated for the presence of the vulnerability. When not suggested otherwise, the subject version is actually the latest release of that version number.

4. **Source Version:** Source version is a term defined in Gumtree as the base version or initial version. It is the version based on which the actions that represent the changes are computed.

5. **Destination Version:** Destination version, as defined in Gumtree, is the second version. It is the second version based on which the actions that represent the changes are computed, i.e. actions are computed from the source version to the destination version.

6. **Node Changes:** Node Changes are objects that store information about the change that happened to the node, i.e. the action that happened to it, identifying information about the node and information about the surrounding context, such as parent node and children. When we use the term *node* with no surrounding context, we are referring to the subject node of the change, or the main node. When we use the term *parent node* we are referring to the node that contains the subject node in its immediate children array. The identifying information about the nodes, referred to as features of Node Changes, depending on the circumstances and action, could be:

    - Action: one of the four actions that can happen to a node*insert*, *insert*, *move*, or *delete*.
    - Node Type (NT): One part of the definition of the node, namely the type of the code element that is being changed, e.g. NumberLiteral.

- Node Label (NL): The second part of the definition of the node, namely the value of the code element that is being changed, e.g. 0.
- Parent Node Type (PT): One part of the definition of the parent node, namely the type of the code element that contains the subject node in its immediate children array, e.g. MethodDeclaration.
- Parent Node Label (PL): The second part of the definition of the parent node, namely the value of the code element that contains the subject node in its immediate children array, e.g. calculateWages().
- New Value (for *update* actions only): the updated/new value of the node, e.g. 1. Essentially, it is a new Node Label.
- ID: the location of a node in the AST, e.g. 12.
- Children: the array of the immediate children of a node, e.g. [TypeDeclaration, ClassDeclaration].

7. **AST Diff Output (ADO):** ADO represents the differencing output between any two versions(source version and destination version) of the same product in an AST format. In our approach, ADO is the differencing output between the vulnerable version V and a subject version X.

8. **Patch AST Representation (PAR):** PAR represents the ADO between the vulnerable version V and fixed version F. It is one of the benchmark artifacts in our approach.

9. **Origin Nodes:** Origin nodes are defined as the nodes that initially contained or introduced the vulnerability.

3.2 Approach

Our goal in this work was to leverage information from the patch that fixed a vulnerability to detect the presence of that said vulnerability across releases. Our approach to reach that goal was to represent the patch in an AST format, and then look for vulnerable and fixing nodes in different versions in order to reach a conclusion about the vulnerability. At the same time, our approach relied in generating artifacts that highlight for the users which nodes are vulnerable and which versions contain them. The conclusion combined with the artifacts could provide a holistic approach to detecting the presence a vulnerability, providing a reasoning for the presence of that vulnerability and guiding developers to fix the vulnerable versions. While our goal was to detect the presence or lack of the vulnerability, our primary focus was to not let any vulnerable versions go undetected by our approach. Considering that we are dealing with an important security issue, it is safer to increase the rate of false positives than the opposite.

To that end, our approach was seeking to categorize versions of a software product in three categories with respect to a known vulnerability. The first category contains those versions that do not have any resemblance to the vulnerable, neither fixed version. These are versions that are not vulnerable: they did not contain the vulnerable nodes in the first place and as such, did

not need to fix them. In the second category are those versions that have significant similarities to the vulnerable version, but not so much to the fixed version. These are versions that might be vulnerable, as they contain vulnerable nodes but not fixes to those vulnerable nodes. Lastly, the third category contains those versions that bear some similarities with the vulnerable version, but even more to the fixed version. These versions are the ones that have higher chances of being not vulnerable, as they might contain traces of vulnerable nodes, but they also have the fixes for them. Having said that, we designed our approach to input a subject version through several checks, with strict thresholds, that would yield a reasonable conclusion pertaining to one of these categories.

The core of the developed methodology revolves around similarities between the PAR and the ADO between *the vulnerable version V* and *a subject version X*.

Considering this and our goal to categorize the versions in three categories, our work is based on the following assertions:

- If the ADO and the PAR have high similarities, it is highly probable that the subject version X is **not vulnerable**.
- If the subject version X is indeed **vulnerable**, its ADO and the relevant PAR would have little to no similarities.

These two cases set the spectrum of decision options for our approach. On one side of the spectrum, we have cases with almost perfect similarities between

the PAR and a subject version's ADO. These cases are the ones in which it is highly likely that the version is indeed not vulnerable. On the other side of the spectrum, we have the cases which are indeed vulnerable, and where the similarity would be very low. Based on this spectrum, all the versions with almost perfect similarity will be considered as not vulnerable. However, defining the scenarios for cases on the other hand of the spectrum is a bit more complicated. Low similarity cases encompass situations where the version is indeed vulnerable, but they can also encompass situations where the version does not contain the elements that introduced the vulnerability. In order to ensure that both of these scenarios are addressed in our approach, we introduced the concept of *origin nodes*. Origin nodes are nodes that introduced or contained the vulnerability in the first place. We identify these nodes through a list of several heuristics that will be explained in details in section 3.3. If the similarity check suggests that the similarity of the ADO and PAR is less than a threshold, the *subject version* is submitted to another process that checks for the presence of the origin nodes in that version. If then, the origin nodes check locates origin nodes in the *subject version*, an additional check is performed to see if those nodes were addressed in the differencing output, if the similarity score is more than 0. If they have not been addressed, we can conclude that the subject version X is vulnerable, as it contains the node and it does not address it. Figure 1 summarizes the methodology.

In general, our approach relies on fine-grained changes between nodes in an AST. Moreover, there are multiple checks the subject version is submitted

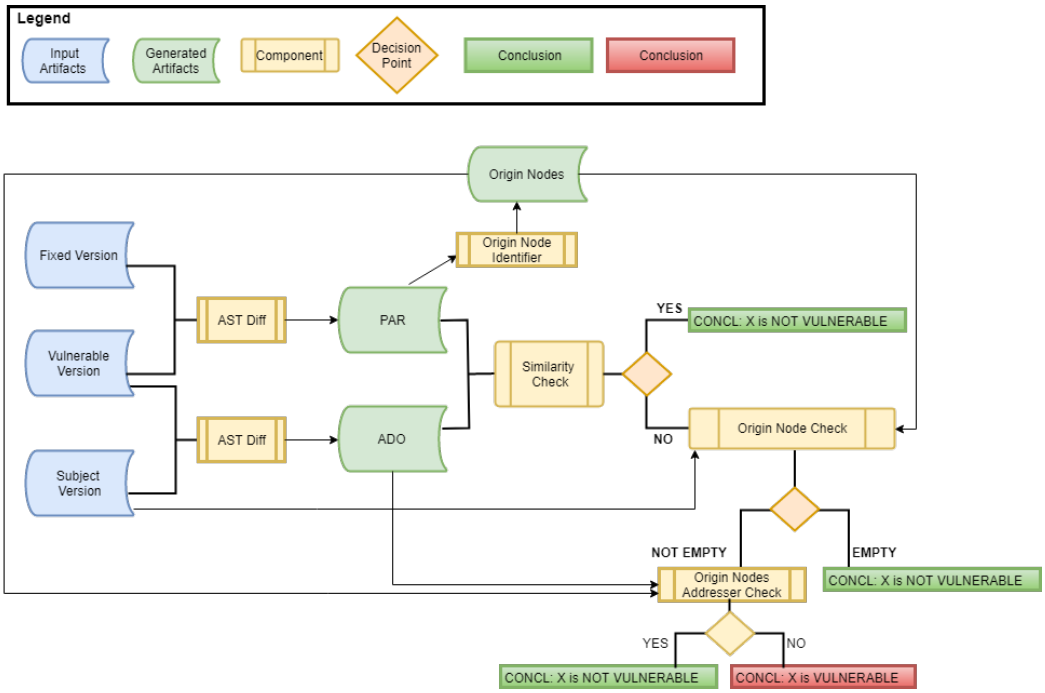

Figure 3.1: Summary of the Approach

to under very high thresholds. This level of strictness is this high to ensure that our approach is concise and especially to not miss-tag versions that are indeed vulnerable. In the following sections, the steps of our approach are described in detail.

3.2.1 AST differencing: PAR and ADO

In our approach, we make use of the Gumtree tool to generate the changes between any two versions [3]. Gumtree takes as input two versions of a software product, i.e. a source version and a destination version, and outputs the AST node changes from the source version to the destination version. The output contains the nodes that have been *inserted*, *moved*, *deleted*, or *updated*.

The nodes themselves are identified by their type, value or label, ID in the AST, and sometimes their parent node, its value or label, and ID as well, and the index in the parent's node children array. As shown in the figure 3.2, the output produces the changes from the source version to the destination version, including the ID number of the node(3 and 16) in the tree and the parent type represented as an integer(55 and 32).

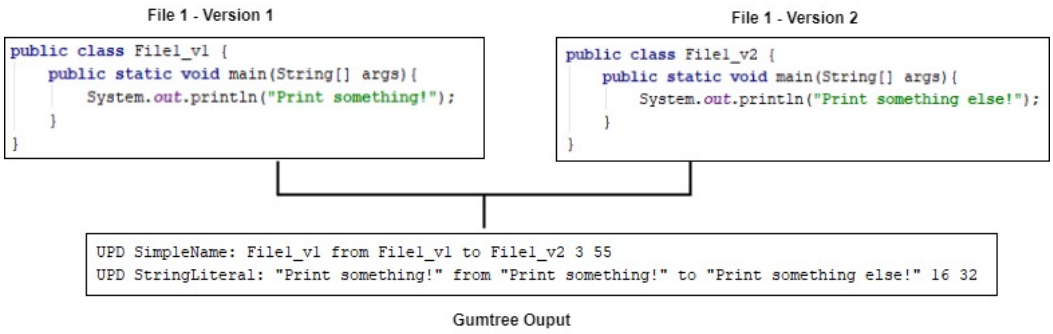

Figure 3.2: Gumtree Output Example

It should be noted that for different actions, the level of information is different. For instance for *move* and *insert* actions, information about the parent node is always present, as the nodes are always inserted or moved to an existing node. However, for *update* and *delete* actions, information about the parent node is not present by default.

In the fist step of our approach, the PAR has to be generated. The PAR, ultimately, is the set of all changes from the vulnerable version to the fixed version presented as changes in AST nodes. PAR is a crucial artifact in our approach as it serves as the benchmark through which the presence of the vulnerability in the other versions is traced. PARs are unique to one particular

CVE, given the fine-level of granularity of changes considered in this approach.

Similarly, the changes in AST from the *vulnerable version* to the *subject version* are also generated in the artifact called ADO. Each ADO has an identifier that relates it to the subject version it belongs to. The ADO is compared against the PAR for similarities, and as such, is part of the evidence to reach the conclusion for the presence of the vulnerability in the subject version it belongs to.

3.2.2 Origin Nodes Identification

Origin nodes are those nodes that introduced the vulnerability in the first place. In order to identify these nodes, we leveraged the information in the PAR and filtered it with several heuristics presented below. Because of different characteristics of the different actions performed on the nodes, the heuristics had to be characterized based on the actions.

1. Move Actions: If the action representing the change in the node is *move*, that means that one node had to be moved to another node. In those cases, considering the general strict level of our approach, we assume that the vulnerability was introduced due to the fact that the node was not in the parent node, or said differently, the parent node did not contain the node. This assumption leads us to consider both the node that was moved and the new parent node as nodes that introduced the vulnerability or origin nodes.

2. Delete and Update Actions: If the action representing the change in the node is either *delete* or *update*, that means that that node was vulnerable and needed to be deleted or updated to fix the vulnerability. In these cases, these nodes are considered origin nodes.

3. Insert Actions: If the action representing the change in the node is *insert*, that means that the parent node needed to have an inserted node for it to be fixed. As such, the parent node was vulnerable. However, based on the format of the PAR, an additional check has to be made for *insert* actions. Usually, the PAR can contain multiple *insert* actions that show how multiple nodes have been inserted to different parent nodes. Moreover, nodes that have been inserted themselves through the patch, can become parent nodes for other nodes that are inserted in them. In these cases, all the parent nodes that were themselves inserted through the patch are discarded and not considered as origin nodes. This check makes sure that these parent nodes are the ones that existed in the vulnerable version, and not introduced as part of the patch. Those parent nodes that pass the check are considered origin nodes.

4. Compilation Unit and Import Declaration Nodes: Lastly, the Compilation Unit nodes, that refer to the file, and Import Declaration nodes, that refer to import statements, do not bear significant information for the presence of the vulnerability. As such, they are ignored from the origin nodes identification.

Algorithm 1 Origin Node Identifier Algorithm

1: **Input:** $NodeChanges \leftarrow \{\forall(NodeChange \in PAR)\}$
2: **Output:** $OriginNodes \leftarrow \{\}$
3: **while** NodeChanges textbfis not empty do **do**
4: **if** NodeChange.Action **is** INS **or** MOV **then**
5: **if** NodeChange.PN **is** ImpDecl **or** ComplUnit **then**
6: $NodeChanges = NodeChanges \setminus NodeChange$
7: **if** NodeChange.Action **is** DEL or UPD or MOV **then**
8: **if** NodeChange.Node **is** ImpDecl **or** ComplUnit **then**
9: $NodeChanges = NodeChanges \setminus NodeChange$
10: **while** NodeChanges **is not** empty **do**
11: **if** NodeChange.Action **is** INS **then**
12: **if** $\neg\exists NodeChange.PN \in \{\forall NodeChanges.Node$ **if** $NodeChanges.Action$ **is** $INS\}$ **then**
13: $OriginNodes \leftarrow NodeChange.PN$
14: **else if** NodeChange.Action **is** DEL or UPD **then**
15: $OriginNodes \leftarrow NodeChange.Node$
16: **else if** NodeChange.Action **is** MOV **then**
17: $OriginNodes \leftarrow NodeChange.Node$
18: $OriginNodes \leftarrow NodeChange.PN$
19: **return** $OriginNodes$

Algorithm 1 details the steps that our approach follows to implement those heuristics. Initially, all the Node Change objects in PAR are added to a set. Node Change objects involving Import Declaration and Compilation Unit nodes are discarded as seen in steps 3-9, respecting the fourth heuristic. Since, for *insert* actions, the parent node is considered as an origin node, the value of the parent node is considered in these steps for these actions. For *update* and *delete* actions, the node itself is considered an origin node, and as such it is the value of the node that is considered in these steps. Lastly, for *move* actions both the node and the parent node are considered origin nodes, so the values of both are considered in these steps for the *move* action. In steps, 6 and 9 the set of Node Changes is updated by removing the nodes that do not pass this check.

The updated set is re-iterated again in step 10. This time, the nodes are added to the Origin Nodes set when they respect heuristics 1-3. In step 12, our approach checks if the parent node of an *insert* Node Change exists in the set of all *insert* Node Changes nodes. This is equivalent to picking only those parent nodes that have not been inserted by the same patch, but rather should have existed in the vulnerable version. In the end, the algorithm returns a set, which holds only unique elements, of origin nodes.

The set of origin nodes is used in comparing the nodes found in ASTs of subject version(s). The origin nodes are stored in an artifact that is unique for each vulnerability. Each artifact has an identifier that relates it to its CVE.

3.2.3 Similarity Check

After the artifacts have been generated, the next step is checking the PAR and each ADO for similarity. This process takes in the PAR and the ADO between the vulnerable and the subject version as input. It tries to match the Node Changes from the PAR to ADO. The matching is performed across different dimensions for different actions.

If all Node Changes are matched, the similarity is 100%. The percentage of similarity is the number of nodes matched over the total number of nodes in the PAR. The similarity score returned from this step is compared to the threshold/s set for similarity.

Algorithm 2 Similarity Check Algorithm

Input: $PARNodeChanges \leftarrow \{\forall(NodeChange \in PAR)\}$
Input: $ADONodeChanges \leftarrow \{\forall(NodeChange \in ADO)\}$
Output: $simscore$
$SimilarNodes \leftarrow \{PARNodeChanges \cap ADONodeChange\}$
$simscore = |PARNodeChanges| \backslash |SimilarNodes|$
return $simscore$

3.2.4 Origin Nodes Check

If the similarity check results in a similarity less than a threshold, the next step is checking for the presence of origin nodes. If the subject version does not contain the origin nodes, this is sufficient evidence to say that the vulnerability did not exist in it. To perform this step, the subject version's file or files involved in the vulnerability are parsed into their AST representation using Gumtree [3]. After the parsing of these files, the program searches for

the origin nodes identified from step II. While searching for the origin nodes, the program keeps a counter. If the counter is 0, the program concludes that the subject version is not vulnerable, and any similarity in the process was due to changes not related to the vulnerability per se. If the counter is higher than 0, the subject version is submitted to an additional check.

3.2.5 Origin Nodes Addresser Check

In the last step, the program checks if the origin nodes are addressed in the similar nodes between the PAR and the ADO. If the counter from origin nodes returns a number different from 0, this is indicative that there is one or more origin nodes present in the subject version. However, the mere presence of those nodes does not indicate that the vulnerability still exists.

For instance, let's assume that a vulnerable node was an *if condition* that was checking if the value of a variable A was 0. Let's assume that the fix involved adding an additional check for the value of another variable B, in the same if condition. If we parse the subject version X, and if we see that the *if condition* checking for the value 0 of variable A is there, its counter for origin nodes will increase. However, in the similar Node Changes between the ADO of subject version X and the relevant PAR, there could be a Node Change object that suggests that the check for the variable B has been added. Our approach needs to evaluate if the identified origin nodes are addressed in the ADO or not. That is why the last step of the approach checks if the origin nodes are among the nodes in similar Node Changes. The program looks

through all origin nodes that the subject version contains and compares them against the nodes found in the similar Node Changes check. If the origin nodes are addressed with the fix, it means that the vulnerability has been addressed and the version is not vulnerable. Otherwise, the origin nodes have not been addressed and the version is vulnerable.

3.2.6 Multiple-File Changes

The patches that fix a vulnerability often times involve more than one file. In cases where multiple-file changes are involved, our approach evaluates all of the files individually. Each file involved in the change has a PAR and Origin Nodes set. All of the subject versions' files are compared against these two artifacts. The subject version is considered not vulnerable if all of the files are evaluated to be non vulnerable; otherwise, if at least one file is considered vulnerable, the version is considered to be vulnerable.

3.2.7 The Five Versions of Patchilyzer

To get a better understanding on what type of information is needed to trace a vulnerability better, we implemented our approach in a tool named Patchilyzer. In deciding how much information to consider when searching and identifying origin nodes, we experimented with five different levels of information. Through this, we wanted to see if there was any correlation between more information regarding origin nodes and higher accuracy. We considered Node Type, Node Label, Parent Node Type, Parent Node Label, and Children

V.	Definition	Example
I	NT + PT + Location	TypeDeclaration 32 190
II	NT + Children	TypeDeclaration [43@@Map]
III	NT + NL + Children	TypeDeclaration int [43@@Map]
IV	NT + NL + PT + Children	TypeDeclaration int 32 [43@@Map]
V	NT + NL + PT + PL + Children	TypeDeclaration int 32 Block [43@@Map]

(3.1)

Table 3.1: The Five Versions of the Patchilyzer

as valuable points of information. The five different versions of Patchilyzer are presented in table 3.1.

The differences among five versions are also present in the PAR and ADO. Because the origin nodes stem from it, the information present in the PAR has to be compatible with the definition of each version for its origin nodes. Because ADO has to be compared to the PAR, its format is conditioned upon PAR by default. For instance, in Patchilyzer I, the PAR and the ADO, always contain information about the Parent Node Type, but not for the node's Children. However, in Patchilyzer II the PAR and the ADO do contain information about the Children as well. Moreover, these differences are reflected on how the five Patchilyzer versions parse the subject version files when doing the Origin Nodes Check, too. Since in this step, the program looks for the origin nodes in the parsed files of the subject version, the parsing format needs to be compatible with the origin nodes format. That is why, for each different version, the parsing follows a different format.

3.3 Experiments

In order to evaluate our approach, we implemented it in the Patchilyzer tool in Java and tested it in 174 versions of Tomcat across 39 vulnerabilities, i.e. CVEs that were found and fixed in Tomcat. Apache Tomcat is an open source web server written in Java, that is used in numerous large applications[1].We chose Tomcat as our case study because it is open source, with numerous vulnerabilities identified and fixed throughout the years. This enabled us to easily access and analyze its source code and the fixes for vulnerabilities. The Tomcat source code is managed in Subversion. However, it also has a mirror repository on github[2]. For the purposes of this experiment, we used Tomcat github repository to collect information about the CVE-s and their fixes. The experiments were carried out in Windows 10, in a 64 bit machine with 3.60 GHz Intel Core CPU.

In this section, we explain the process of gathering the data, the preprocessing, and the evaluation process.

3.3.1 Gathering Data and Preprocessing

For our approach to work, we needed to collect several data points from the Tomcat repository. First, we needed to obtain all the CVE-s that were fixed. To do that, we followed some heuristics. Usually, when developers fix a vulnerability, they refer to the CVE they fixed in the commit message as

[1]http://tomcat.apache.org/
[2]https://github.com/apache/tomcat/

seen in Figure 3.3. Using this piece of information, we ran a crawler in the Tomcat github mirror repository, to search free-text on the commit message and identify a CVE-XXX-XXX regex pattern. We also needed to get the commits that fixed those patches. Based on our CVE search, we obtained the hash of the resulting commit, i.e. the fix version, the previous commit, i.e. the vulnerable version, and the files changed in the fixed version. After

Figure 3.3: Tomcat CVE Fix Commit Message Example

the information had been obtained, a semi-automatic processing of the data ensued. First, all non-source code files were discarded, including test files, xml files or change logs. Second, in our dataset, we had cases where a CVE was fixed by two or more consecutive commits. In these cases, the definition for the vulnerable version and fixed version changes. Specifically, the revision were vulnerability was found, i.e. the vulnerable version, is the one previous to the earliest commit, and the revision where the vulnerability was fixed is the most recent one where that CVE was mentioned. Since our approach relies on leveraging all the information that fixed a vulnerability, we had to combine the patches in cases like this to get the full picture of all the changes that happened. The only way to do that is adding up and combining the changes from the two patches. Therefore, in cases with multiple fix-commits, our vulnerable version was the revision previous to the first time that CVE

was mentioned and the fixed version was the last revision where the CVE was mentioned. To address this issue, we merged the cases with the CVEs from consecutive commits. As such, we essentially combined the two patches together in one.

To illustrate this process with an hypothetical example, as seen in Figure 3.4, let's assume that CVE-AAA-AAA was fixed in two consecutive commits, namely commit 2 and commit 3. In the second instance this CVE is mentioned, the fix commit from the previous time, i.e. commit 2, became the vulnerable version according to our initial definition. However, this is not the case for this CVE. The vulnerable version is commit 1. In order to evaluate all of the changes done to fix the vulnerability, we need to combine all changes from commit 1 to commit 2, and then all the changes from commit 2 to commit 3. The easiest way to do this, is look at all the changes from commit 1, the vulnerable version, to commit 3, the final fixed version. The changes reported from commit 1 to commit 3 will contain the changes done in commit 2 as well. As such, in this case, we considered the revision previous to the revision with the earliest mention of the CVE id, i.e. commit 1, as our vulnerable version. Similarly, the last commit, i.e. commit 3, where the CVE was mentioned is considered as the fixed commit.

There were also cases where the same CVE was fixed from non-consecutive commits. In those cases we evaluated the subject version for all of the consecutive commits individually, and the final conclusion regarding the version was an aggregate of all the individual decisions.

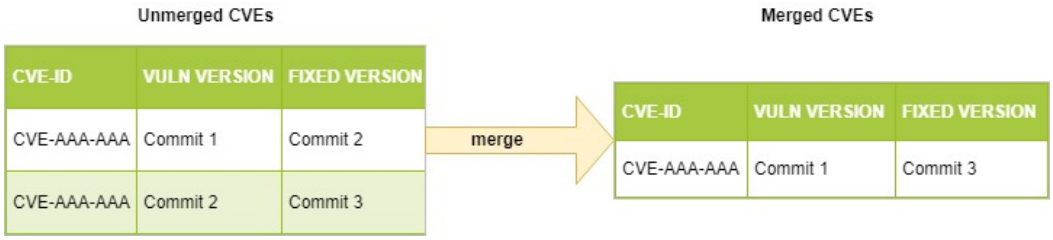

Figure 3.4: CVE Merging Example

Simultaneously, the source code of all supported versions of Tomcat was cloned in the local machine where we were carrying our experiments.

Finally, the paths to the vulnerable and fixed versions of 39 CVEs, the names of the files that changed in the commit, and the directory with the 174 subject versions to be evaluated was given as input to Patchilyzer. This was the data used to perform the experiments. The experiments were carried out across the five versions of Patchilyzer from section 3.2.7. Moreover, they were carried out with three different thresholds for similarity check, namely 0.7, 0.8, and 0.9.

3.3.2 Evaluation Process

In the end, after the experiments were carried out, we had a total of 6,786 combinations of CVEs and versions that had been evaluated by our approach. In order to evaluate the accuracy of our approach, going through all these combinations would have been a very tedious and time-consuming task. However, there were some ways in which we could remove some of the combinations automatically. There were cases where the similarity score was one, and there were also cases where the similarity score was 0, with 0 origin

nodes found in the subject version. In both of these cases, our confidence that the subject version is not vulnerable is higher. Moreover, there were cases where the files that were changed as part of a vulnerability fix did not exist at all in the subject version. All of these cases were scenarios in which we a manual analysis was not highly necessary. As such they were discarded.

What we needed to more urgently was evaluate cases that were in the middle part of the decision spectrum, namely cases with similarities from 0 with some original nodes present, inclusive, to less than 1, non-inclusive. These were the cases where the presence of the vulnerability was not straightforward. As such, we ended up with a combination of around 3,000 CVEs and versions to be evaluated. The amount of data was still high and required a lot of manual work to be done. Ultimately, we chose a random sample of 35 combinations of CVEs and versions to be manually evaluated by four reviewers with relevant background. The four reviewers were given all the information they needed, i.e. the vulnerable version, the fixed version, the patch and the subject version, and they were asked to perform manual analyses on the source code of the 35 subject versions to try to pinpoint if a particular vulnerability was still located there. The reviewers were asked to provide a reasoning for their conclusion. The reasoning was thoroughly manually vetted for validity. If issues were encountered, the reviewers were asked to re-do the review, until it was considered valid and contained sufficient evidence to pass the vetting.

Chapter 4

Results and Discussion

Our approach was evaluated using Tomcat's source code and its vulnerabilities. For our experiments, we used three different thresholds, specifically, 0.7, 0.8, and 0.9, for the similarity check. The thresholds were set at this high level because one of our goals was to have a strict approach when it comes to evaluating the similarity between PAR and ADO. Moreover, our origin node threshold was set to at least 1, in all the cases, i.e. even if one origin node existed that was not addressed, the version was tagged as vulnerable. It was our primary goal and focus to not miss-tag vulnerable versions for not vulnerable and our thresholds reflect this goal. Lastly, the five different versions of Patchilyzer were used when running the experiment; as such, their accuracy was evaluated as well. Through the five different versions, we wanted to discern any significant changes in accuracy level with differing features in origin nodes definition.

After obtaining the evaluation results from our reviewers, we calculated the accuracy of our approach for every threshold and every version of Patchilyzer. The results are summarized in Table 4.1. As it can be seen from the table, Patchilyzer III demonstrates higher accuracy than all other versions,

V./ Thresholds	**0.7**	**0.8**	**0.9**
Patchilyzer I	0.8	0.77	0.66
Patchilyzer II	0.83	0.74	0.57
Patchilyer III	0.86	0.8	0.69
Patchilyzer IV	0.83	0.74	0.63
Patchilyzer V	0.86	0.78	0.66

(4.1)

Table 4.1: Accuracy Results for Patchilzyer's Versions and Thresholds

under all thresholds, with 86% being its highest accuracy. Patchilyzer V has a similar accuracy level, but for thresholds 0.7 and 0.9 its accuracy lowers compared to Patchilyzer III. Patchilyzers II and IV have similar accuracy levels as well, besides on the 0.9 threshold, where Patchiylzer IV has a slightly higher accuracy. Lastly, Patchilyzer I performs worse than all of the other versions. In section 4.1., we dwell into the rationale behind these results.

The different thresholds also have varying accuracy levels. However, the ranking is the same for all of the cases. Threshold 7.0 has the highest accuracy levels, followed by 0.8 and 0.9. A discussion of this perspective on the results is presented in section 4.2.

4.1 Versions' Comparison

The results reveal an interesting picture. First, in terms of the five versions of Patchilyzer, the results show that while up to some point having more information on origin nodes increases the accuracy, this is not always the case. If we compare Patchilyzer III with Patchilyzers I and II, that use less

information in how they define origin nodes, we see that the accuracy always increases for Patchilyzer III.

Regarding Patchilyzer I, one rationale behind its performance is that it uses the location or ID of the node as one of the defining features for origin nodes. Since the overall AST of a file might have changes that are not related to the vulnerability at all, the location of the nodes that are indeed involved in the change is impacted because of this. As such, due to the non-relevant changes in the ID, this comparison point, i.e. the location, did not match for instances when they should have matched, or it accidentally matched when it should not have.

Let's take for instance CVE-2010-4172. For both Patchilyzer I and Patchilyzer III, the similarity score was below all the thresholds. Patchilyzer I detected that Tomcat 7-0-4 did not contain this vulnerability, whereas Patchilyzer III detected that it did. When we look into the artifacts produced by both of the approaches, we can see that Patchilyzer I, in the first file that was changed in the patch, identified three origin nodes. That means that, per the approach taken by Patchilyzer I, there were three different nodes that contained or introduced the vulnerability. Patchilyzer III also identified three origin nodes, but the information it used to define those origin nodes was slightly different. When Patchilyzer I looked for the origin nodes in the parsed file of the subject version, it couldn't identify any. However, Patchilyzer III did in fact find one origin nodes in the file, a Return Statement. Moreover, it found that that node was not addressed. When we look at the source code,

Version	Sim. Score	Add. ON/Prs. ON	Conclusion
Patchilyzer I	0.33	0/0	NOT VULN
Patchilyzer III	0.33	0/1	VULN

Table 4.2: Evaluation of Tomcat 7-0-4 for CVE-2010-4172

the origin node, that Return Statement, was indeed there, but due to other changes, it ended up in a different location in the AST and Patchilyzer I was not able to catch it. As such, Patchilyzer I ended up miss-tagging Tomcat 7-0-14 as not vulnerable, when it really was. Cases like this have negatively impacted the accuracy of Patchilyzer I. The case study of Patchilyzer I suggests that the location might not be a good comparison point when trying to identify origin nodes.

Patchilyzer II, on the other hand, does not rely on the ID of the node, but on the Node Label and Children. By looking at the data, it seems like for the cases that Patchilyzer III detected correctly that were missed by Patchilyzer II, the problem was that origin nodes were defined loosely in Patchilyzer II. Because of this, there was a higher probability for each identified origin node to be located in the parsed AST of the subject version. If an origin node was simply defined as Block, with no children, there were higher chances of a Block code element to exist in the parsed AST of the subject version. This increased the rate of matching origin nodes, contributing in false positives. Patchilyzer III fixes this problem by adding one more point of comparison that helps increase the accuracy.

A case like this happened with a partial fix for CVE-2011-2526. For

Version	Sim. Score	Add. ON/Prs. ON	Conclusion
Patchilyzer II	0.67	2/3	VULN
Patchilyzer III	0.33	1/1	NOT VULN
Patchilyzer IV	0.33	0/1	VULN

Table 4.3: Evaluation of Tomcat 7-0-4 for CVE-2010-4172

both Patchilyzers II and III, the similarity score was below all the thresholds. Patchilyzer II identified three origin nodes in that fix, and it also identified their presence in the subject version, Tomcat 8-5-8. Moreover, it found out that only one of those origin nodes was actually addressed in the similar nodes. As such, it concluded that this version did contain the vulnerability. Patchilyzer III, on the other hand, also identified three origin nodes, but it found only one in the same subject version. That same origin node had been addressed and fixed, and Patchilyzer III concluded that the version was not vulnerable. The version was indeed not vulnerable. What had happened is that because the parsing in Patchilyzer II is vaguer than in Patchilyzer III, Patchilyzer II had located other false-positive nodes with similar characteristics as the origin nodes that were not really vulnerable. Because of the additional information required by Patchilyzer III, it was able to decrease the vagueness in this case.

However, if, on the other hand, we look at the results of Patchilyzers IV compared to Patchilyzer III, we see that the accuracy level does not increase, and it even slightly decreases. This information suggests that more information does not necessarily increase the detection accuracy. Why is that? For starters, the level of information provided in Patchilyzer III can be the level in which

our approach reaches information saturation. Any piece of information after that does not add to the detection accuracy, but just strengthens it. However, this answer does not address the one case in which the conclusion reached by Patchilyzers III and V differs from that achieved by Patchilyzer IV, namely CVE-2011-2526. In this case, the similarity scores were below the thresholds for Patchilyzer III and IV, but the problem was presented after the origin nodes check. Because Patchilyzer III allowed for more flexibility in the origin nodes, it was able to detect better the fixed origin nodes. However, the stricter approach of Patchilyzer IV, which was looking in the similarity nodes for exact matches of their more complex origin nodes, failed to detect the fix which was a bit different.

In conclusion, the cross comparison of the version suggests that up to a point, more information in defining origin nodes increases the accuracy; however, after a certain point that accuracy does not improve, and it might actually be harmed.

4.2 Thresholds' Comparison

The threshold results paint, to some degree, the same picture as the version comparison. Nonetheless, the threshold comparison gives us another perspective since, differently from versions that alter the origin node check, they are set for the similarity check. Because of our spectrum categorization, and because we were looking for cases with almost perfect similarities, we set the thresholds for similarity very high. However, looking at the data, if

our thresholds were set lower, the detection accuracy would have decreased for thresholds lower than 0.7. Nonetheless, despite this, we also see that on average, the lowest of our thresholds, 0.7, performs better than the other two. Moreover, the 0.9 thresholds, which is even closer to perfect similarity, always performs the worse out of the three. In the Patchilyzer II example, the decrease in accuracy from the 0.8 to 0.9 thresholds is for 23%. The reasoning behind this has again to do with how the fixes would look like from one version to another. The more dimensions added to the matching criteria between the Node Changes objects, the less realistic the detection becomes, as it assumes that all the versions will have the same fix as the fixed version. However, the fixes from one version to another might have similar elements, but also might adjust to accommodate other changes in that version. That is why the threshold at 0.7 allows for a more realistic representation of the fix, which enables it to perform better in terms of accuracy.

4.3 Common Threads and Validity of Our Approach

Across all the versions and thresholds, there were some common cases that give us better insights on how the overall approach performs. Apart from Patchilyzer I, all of other Patchilyzers in all of the thresholds successfully identified the versions that were indeed vulnerable. In other words, in all of the cases were reviewers said a version was vulnerable, Patchilyzers II to V accurately identified them as vulnerable as well. This reflects that our strict approach for the aim of not missing true vulnerable versions was successful.

On the other hand, in all of the cases in which Patchilyzers II to V did not agree with the reviewers was on cases where the vulnerability was not there, but our approach tagged them as vulnerable. In other words, all of the false positives for Patchilyzer II, III, IV, and V involve mis-tagging non-vulnerable versions for vulnerable. In terms of our overall goals, this is a trade-off between loosening our thresholds and risking to miss true vulnerable versions and this scenario in which we miss-tag non-vulnerable versions for vulnerable. The present scenario, in which we keep our thresholds tight and miss-tag non-vulnerable versions for vulnerable, is a trade-off scenario which serves the purpose of this work better. In terms of achieving security, it is better to be more cautious than less cautious. Moreover, cases like the ones our approach have mis-tagged, that detect some similarity between the PAR and the ADO, could point to partial fixes or to instances that require the developers' attention to be completely fixed.

4.4 Results from Previous Work

There were two previous research studies that focused on tracing vulnerabilities across different releases. As mentioned above, their focus is lines of code changed, i.e. added or deleted, to fix a vulnerability. Comparing the accuracy of these tools to ours is a bit tricky, as they have used different datasets and number of conclusions. For instance, in [1], the dataset included several CVEs from Apache Hadoop, Cloudstack and HTTP Server. In their approach, they used different thresholds, namely 0.5 for additions and 0.25 for deletions.

The way they used the thresholds is as follows: if the subject version contains less than 0.5 of additions or more than 0.25 of deletions, a version is considered vulnerable. Moreover, they made use of three different conclusions for their evaluation set: *vulnerable*, *not vulnerable*, or *indeterminate*. After removing the *indeterminate* results from the dataset, for the remaining of the data, the reported accuracy level was 86%.

In another work of this nature, the presence of deleted lines in a patch was considered as what makes a subject version vulnerable [13]. The researchers obtained what is called the vulnerability footprint, containing the deleted lines in the patch, and looked for its presence back in other releases. When lines of code were only added to fix a vulnerability, the entire file where these lines of coded were added was considered as vulnerable. To evaluate the accuracy of their approach, the researchers tested their approach in 80 vulnerabilities from Firefox, and then manually evaluated those. The accuracy level was 80%.

Comparing these data to our evaluation process cannot be done directly, as in all three works the evaluation datasets are different from one another. Moreover, in [1], the types of conclusions the approach can reach differ from our approach, as they include the indeterminate result. As such, it's best to interpret these results on all the different approaches against the datasets used in their respective work.

4.5 Threats to Validity

The work presented in this thesis is subject to some threats to validity.

First, the final results excluded the versions that had a similarity score of 1, and those that had a similarity score 0. This was done to focus on the most challenging section of the spectrum of our results; however, this prohibits us from displaying the full tracing power of the tool. Further analysis done for cases like this needs to be conducted to eliminate this threat.

Second, the cases in which files involved in the patch did not exist in the subject versions were not looked into further to consider changes in the filenames or directories. This might have affected the final results and an approach that traces these files back to older versions needs to be implemented as a complimentary approach to the Patchilyzer.

Lastly, although the reviewers were asked to provide a rationale for each of their decisions and were given instructions, a firmer conclusion could be reached if exploits of these CVEs would be ran against the source code. Such an approach could be used in the future.

Chapter 5

Conclusion and Future Work

5.1 Future Work

Future work in this area can initially focus in testing the approach in products written in different programming languages and platforms. As of now, our approach has been tested only in Tomcat. A variety of products and programming languages could analyze how our tool works with other languages and give us better insights on the strengths and drawbacks. Moreover, statistical models and machine learning algorithms could be used to give weights to origin nodes that better detect the presence of a vulnerability. The ground-truth that could be used for such a project could be the dataset build in this work coupled with the reviewers' evaluation. Based on theoretical knowledge, we already discarded too broad origin nodes such as CompilationUnit, or the ones that do not bear significant information to the functionality of the software product, such as ImportDeclaration. However, such a project, involving machine learning, could give us better insights in how to increase the detection accuracy through giving more weight to certain origin nodes and discarding others. Lastly, future work can be focused towards making the PAR generalizable and abstract away program-specific features, to detect known vulnerabilities across projects.

5.2 Conclusion

Ensuring the security of software products is increasingly taking a central role in the engineering software practices. Due to its importance and complexity, this task becomes time consuming for developers. As such, there is a need to automate processes that could help developers ensure the security of their products with high accuracy. To that end, through this project, we created an approach that relies on AST representation of source code to trace known vulnerabilities across releases of software product. Our work, to the best of our knowledge, is the first of its kind, in that it relies on a more flexible representation of the source code to solve this problem. Previous work relies on lines of code as means of detecting vulnerabilities. Moreover, our approach generates comprehensive artifacts that could help developers with guidance on what and how to patch. Simultaneously, our main aim was to make this approach as strict as possible to not miss-tag any vulnerable versions as non vulnerable, as the opposite would have serious security effects for those who rely on our tool.

We implemented our approach in a Java tool named Patchilyzer. The accuracy of our tool was evaluated through five different versions and three different thresholds. Three reviewers were asked to manually review the accuracy of our tool. We reached an accuracy of 86% in detecting the presence of a vulnerability. The cases in which we mis-tagged the vulnerable status of a subject version where cases where the subject version was not vulnerable. This comes from our approach being strict in that it has high thresholds for versions

to pass not to be considered vulnerable. This reflects our overall tendency not to miss versions that are indeed vulnerable. In our view, miss-tagging some versions as vulnerable, when they are not, is better than the opposite: not warning developers and users that a version is vulnerable when it really is. The risk of having to make this trade-off was the reason behind holding on to stricter thresholds.

Chapter 6

Appendix - Raw Results

In this chpater, we will present the raw results from the evaluation process. The results from each version of Patchilyzer are presented in a separate table. The table contains the conclusion from all of the thresholds used and the conclusion reached by the reviewer.

Table 6.1: Pathilyzer I Raw Results

CVE-ID	Version	0.7	0.8	0.9	Review
CVE-2007-0450	tomcat_7_0_21	NOT VULN	NOT VULN	NOT VULN	VULN
CVE-2007-2450	tomcat_7_0_23	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2008-1947	tomcat_7_0_14	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2008-5515	tomcat_7_0_65	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2009-2693	tomcat_8_0_11	NOT VULN	VULN	VULN	NOT VULN
CVE-2009-2901	tomcat_8_0_1	NOT VULN	NOT VULN	VULN	NOT VULN
CVE-2009-2902	tomcat_8_5_21	NOT VULN	NOT VULN	VULN	NOT VULN
CVE-2009-3555	tomcat_7_0_19	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2009-3555_1	tomcat_8_5_22	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2010-1157	tomcat_7_0_63	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2010-1622	tomcat_8_0_16	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2010-2227_1	tomcat_7_0_12	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2010-3718	tomcat_8_0_49	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2010-4172	tomcat_7_0_4	NOT VULN	NOT VULN	NOT VULN	VULN
CVE-2010-4476	tomcat_7_0_6	NOT VULN	NOT VULN	NOT VULN	VULN
CVE-2011-0013	tomcat_7_0_64	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-0534	tomcat_7_0_26	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-1088	tomcat_7_0_62	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-1088_1	tomcat_7_0_77	NOT VULN	NOT VULN	NOT VULN	VULN
CVE-2011-1088_2	tomcat_7_0_37	NOT VULN	NOT VULN	NOT VULN	VULN
CVE-2011-1183	tomcat_8_0_8	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-1184	tomcat_7_0_71	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-1475	tomcat_9_0_0	NOT VULN	NOT VULN	NOT VULN	VULN
CVE-2011-1582	tomcat_8_0_37	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-2204	tomcat_8_5_20	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-2481	tomcat_7_0_36	NOT VULN	NOT VULN	VULN	NOT VULN
CVE-2011-2481_1	tomcat_7_0_43	NOT VULN	NOT VULN	VULN	NOT VULN
CVE-2011-2526	tomcat_7_0_40	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-2526_1	tomcat_8_5_8	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-2526_2	tomcat_8_0_20	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-2526_3	tomcat_7_0_44	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-3190	tomcat_8_0_41	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2014-0050	tomcat_8_0_36	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2016-5388	tomcat_7_0_49	NOT VULN	NOT VULN	NOT VULN	VULN
CVE-2017-12617	tomcat_7_0_20	NOT VULN	NOT VULN	NOT VULN	NOT VULN

Table 6.2: Patchilyzer II Raw Results

CVE-ID	Version	0.7	0.8	0.9	Review
CVE-2007-0450	tomcat_7_0_21	VULN	VULN	VULN	VULN
CVE-2007-2450	tomcat_7_0_23	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2008-1947	tomcat_7_0_14	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2008-5515	tomcat_7_0_65	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2009-2693	tomcat_8_0_11	NOT VULN	VULN	VULN	NOT VULN
CVE-2009-2901	tomcat_8_0_1	NOT VULN	NOT VULN	VULN	NOT VULN
CVE-2009-2902	tomcat_8_5_21	NOT VULN	VULN	VULN	NOT VULN
CVE-2009-3555	tomcat_7_0_19	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2009-3555_1	tomcat_8_5_22	VULN	VULN	NOT VULN	VULN
CVE-2010-1157	tomcat_7_0_63	NOT VULN	VULN	VULN	NOT VULN
CVE-2010-1622	tomcat_8_0_16	VULN	VULN	VULN	NOT VULN
CVE-2010-2227_1	tomcat_7_0_12	NOT VULN	NOT VULN	VULN	NOT VULN
CVE-2010-3718	tomcat_8_0_49	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2010-4172	tomcat_7_0_4	VULN	VULN	VULN	VULN
CVE-2010-4476	tomcat_7_0_6	VULN	VULN	VULN	VULN
CVE-2011-0013	tomcat_7_0_64	NOT VULN	NOT VULN	VULN	NOT VULN
CVE-2011-0534	tomcat_7_0_26	NOT VULN	NOT VULN	VULN	NOT VULN
CVE-2011-1088	tomcat_7_0_62	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-1088_1	tomcat_7_0_77	VULN	VULN	VULN	VULN
CVE-2011-1088_2	tomcat_7_0_37	VULN	VULN	VULN	VULN
CVE-2011-1183	tomcat_8_0_8	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-1184	tomcat_7_0_71	NOT VULN	NOT VULN	VULN	NOT VULN
CVE-2011-1475	tomcat_9_0_0	VULN	VULN	VULN	VULN
CVE-2011-1582	tomcat_8_0_37	VULN	VULN	VULN	NOT VULN
CVE-2011-2204	tomcat_8_5_20	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-2481	tomcat_7_0_36	VULN	VULN	VULN	NOT VULN
CVE-2011-2481_1	tomcat_7_0_43	VULN	VULN	VULN	NOT VULN
CVE-2011-2526	tomcat_7_0_40	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-2526_1	tomcat_8_5_8	VULN	VULN	VULN	NOT VULN
CVE-2011-2526_2	tomcat_8_0_20	VULN	VULN	VULN	NOT VULN
CVE-2011-2526_3	tomcat_7_0_44	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-3190	tomcat_8_0_41	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2014-0050	tomcat_8_0_36	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2016-5388	tomcat_7_0_49	VULN	VULN	VULN	VULN
CVE-2017-12617	tomcat_7_0_20	NOT VULN	NOT VULN	NOT VULN	NOT VULN

Table 6.3: Patchilyzer III Raw Results

CVE-ID	Version	0.7	0.8	0.9	Review
CVE-2007-0450	tomcat_7_0_21	VULN	VULN	VULN	VULN
CVE-2007-2450	tomcat_7_0_23	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2008-1947	tomcat_7_0_14	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2008-5515	tomcat_7_0_65	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2009-2693	tomcat_8_0_11	NOT VULN	VULN	VULN	NOT VULN
CVE-2009-2901	tomcat_8_0_1	NOT VULN	NOT VULN	VULN	NOT VULN
CVE-2009-2902	tomcat_8_5_21	NOT VULN	VULN	VULN	NOT VULN
CVE-2009-3555	tomcat_7_0_19	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2009-3555_1	tomcat_8_5_22	VULN	VULN	VULN	NOT VULN
CVE-2010-1157	tomcat_7_0_63	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2010-1622	tomcat_8_0_16	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2010-2227_1	tomcat_7_0_12	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2010-3718	tomcat_8_0_49	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2010-4172	tomcat_7_0_4	VULN	VULN	VULN	VULN
CVE-2010-4476	tomcat_7_0_6	VULN	VULN	VULN	VULN
CVE-2011-0013	tomcat_7_0_64	NOT VULN	NOT VULN	VULN	NOT VULN
CVE-2011-0534	tomcat_7_0_26	NOT VULN	NOT VULN	VULN	NOT VULN
CVE-2011-1088	tomcat_7_0_62	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-1088_1	tomcat_7_0_77	VULN	VULN	VULN	VULN
CVE-2011-1088_2	tomcat_7_0_37	VULN	VULN	VULN	VULN
CVE-2011-1183	tomcat_8_0_8	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-1184	tomcat_7_0_71	NOT VULN	NOT VULN	VULN	NOT VULN
CVE-2011-1475	tomcat_9_0_0	VULN	VULN	VULN	VULN
CVE-2011-1582	tomcat_8_0_37	VULN	VULN	VULN	NOT VULN
CVE-2011-2204	tomcat_8_5_20	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-2481	tomcat_7_0_36	VULN	VULN	VULN	NOT VULN
CVE-2011-2481_1	tomcat_7_0_43	VULN	VULN	VULN	NOT VULN
CVE-2011-2526	tomcat_7_0_40	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-2526_1	tomcat_8_5_8	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-2526_2	tomcat_8_0_20	VULN	VULN	VULN	NOT VULN
CVE-2011-2526_3	tomcat_7_0_44	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-3190	tomcat_8_0_41	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2014-0050	tomcat_8_0_36	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2016-5388	tomcat_7_0_49	VULN	VULN	VULN	VULN
CVE-2017-12617	tomcat_7_0_20	NOT VULN	NOT VULN	NOT VULN	NOT VULN

Table 6.4: Patchilyzer IV Raw Results

CVE-ID	Version	0.7	0.8	0.9	Review
CVE-2007-0450	tomcat_7_0_21	VULN	VULN	VULN	VULN
CVE-2007-2450	tomcat_7_0_23	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2008-1947	tomcat_7_0_14	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2008-5515	tomcat_7_0_65	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2009-2693	tomcat_8_0_11	NOT VULN	VULN	VULN	NOT VULN
CVE-2009-2901	tomcat_8_0_1	NOT VULN	NOT VULN	VULN	NOT VULN
CVE-2009-2902	tomcat_8_5_21	NOT VULN	VULN	VULN	NOT VULN
CVE-2009-3555	tomcat_7_0_19	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2009-3555_1	tomcat_8_5_22	VULN	VULN	VULN	NOT VULN
CVE-2010-1157	tomcat_7_0_63	NOT VULN	VULN	VULN	NOT VULN
CVE-2010-1622	tomcat_8_0_16	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2010-2227_1	tomcat_7_0_12	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2010-3718	tomcat_8_0_49	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2010-4172	tomcat_7_0_4	VULN	VULN	VULN	VULN
CVE-2010-4476	tomcat_7_0_6	VULN	VULN	VULN	VULN
CVE-2011-0013	tomcat_7_0_64	NOT VULN	NOT VULN	VULN	NOT VULN
CVE-2011-0534	tomcat_7_0_26	NOT VULN	NOT VULN	VULN	NOT VULN
CVE-2011-1088	tomcat_7_0_62	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-1088_1	tomcat_7_0_77	VULN	VULN	VULN	VULN
CVE-2011-1088_2	tomcat_7_0_37	VULN	VULN	VULN	VULN
CVE-2011-1183	tomcat_8_0_8	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-1184	tomcat_7_0_71	NOT VULN	NOT VULN	VULN	NOT VULN
CVE-2011-1475	tomcat_9_0_0	VULN	VULN	VULN	VULN
CVE-2011-1582	tomcat_8_0_37	VULN	VULN	VULN	NOT VULN
CVE-2011-2204	tomcat_8_5_20	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-2481	tomcat_7_0_36	VULN	VULN	VULN	NOT VULN
CVE-2011-2481_1	tomcat_7_0_43	VULN	VULN	VULN	NOT VULN
CVE-2011-2526	tomcat_7_0_40	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-2526_1	tomcat_8_5_8	VULN	VULN	VULN	NOT VULN
CVE-2011-2526_2	tomcat_8_0_20	VULN	VULN	VULN	NOT VULN
CVE-2011-2526_3	tomcat_7_0_44	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-3190	tomcat_8_0_41	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2014-0050	tomcat_8_0_36	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2016-5388	tomcat_7_0_49	VULN	VULN	VULN	VULN
CVE-2017-12617	tomcat_7_0_20	NOT VULN	NOT VULN	NOT VULN	NOT VULN

Table 6.5: Patchilyzer V Raw Results

CVE-ID	Version	0.7	0.8	0.9	Review
CVE-2007-0450	tomcat_7_0_21	VULN	VULN	VULN	VULN
CVE-2007-2450	tomcat_7_0_23	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2008-1947	tomcat_7_0_14	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2008-5515	tomcat_7_0_65	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2009-2693	tomcat_8_0_11	NOT VULN	VULN	VULN	NOT VULN
CVE-2009-2901	tomcat_8_0_1	NOT VULN	NOT VULN	VULN	NOT VULN
CVE-2009-2902	tomcat_8_5_21	NOT VULN	VULN	VULN	NOT VULN
CVE-2009-3555	tomcat_7_0_19	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2009-3555_1	tomcat_8_5_22	VULN	VULN	VULN	NOT VULN
CVE-2010-1157	tomcat_7_0_63	NOT VULN	VULN	VULN	NOT VULN
CVE-2010-1622	tomcat_8_0_16	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2010-2227_1	tomcat_7_0_12	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2010-3718	tomcat_8_0_49	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2010-4172	tomcat_7_0_4	VULN	VULN	VULN	VULN
CVE-2010-4476	tomcat_7_0_6	VULN	VULN	VULN	VULN
CVE-2011-0013	tomcat_7_0_64	NOT VULN	NOT VULN	VULN	NOT VULN
CVE-2011-0534	tomcat_7_0_26	NOT VULN	NOT VULN	VULN	NOT VULN
CVE-2011-1088	tomcat_7_0_62	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-1088_1	tomcat_7_0_77	VULN	VULN	VULN	VULN
CVE-2011-1088_2	tomcat_7_0_37	VULN	VULN	VULN	VULN
CVE-2011-1183	tomcat_8_0_8	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-1184	tomcat_7_0_71	NOT VULN	NOT VULN	VULN	NOT VULN
CVE-2011-1475	tomcat_9_0_0	VULN	VULN	VULN	VULN
CVE-2011-1582	tomcat_8_0_37	VULN	VULN	VULN	NOT VULN
CVE-2011-2204	tomcat_8_5_20	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-2481	tomcat_7_0_36	VULN	VULN	VULN	NOT VULN
CVE-2011-2481_1	tomcat_7_0_43	VULN	VULN	VULN	NOT VULN
CVE-2011-2526	tomcat_7_0_40	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-2526_1	tomcat_8_5_8	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-2526_2	tomcat_8_0_20	VULN	VULN	VULN	NOT VULN
CVE-2011-2526_3	tomcat_7_0_44	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2011-3190	tomcat_8_0_41	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2014-0050	tomcat_8_0_36	NOT VULN	NOT VULN	NOT VULN	NOT VULN
CVE-2016-5388	tomcat_7_0_49	VULN	VULN	VULN	VULN
CVE-2017-12617	tomcat_7_0_20	NOT VULN	NOT VULN	NOT VULN	NOT VULN

Bibliography

[1] Craig Cabrey. *Identifying the Presence of Known Vulnerabilities in the Versions of a Software Project*. Rochester Institute of Technology, 2016.

[2] Istehad Chowdhury and Mohammad Zulkernine. Using complexity, coupling, and cohesion metrics as early indicators of vulnerabilities. *Journal of Systems Architecture*, 57(3):294–313, 2011.

[3] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 313–324. ACM, 2014.

[4] Andreas Fischer, Ching Y Suen, Volkmar Frinken, Kaspar Riesen, and Horst Bunke. Approximation of graph edit distance based on hausdorff matching. *Pattern Recognition*, 48(2):331–343, 2015.

[5] Steve Hanna, Ling Huang, Edward Wu, Saung Li, Charles Chen, and Dawn Song. Juxtapp: A scalable system for detecting code reuse among android applications. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 62–81. Springer, 2012.

[6] Aram Hovsepyan, Riccardo Scandariato, Wouter Joosen, and James Walden. Software vulnerability prediction using text analysis techniques. In *Proceedings of the 4th international workshop on Security measurements and metrics*, pages 7–10. ACM, 2012.

[7] Jiyong Jang, Abeer Agrawal, and David Brumley. Redebug: finding unpatched code clones in entire os distributions. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 48–62. IEEE, 2012.

[8] Jinyoo Kim, Yashwant K Malaiya, and Indrakshi Ray. Vulnerability discovery in multi-version software systems. In *High Assurance Systems Engineering Symposium, 2007. HASE'07. 10th IEEE*, pages 141–148. IEEE, 2007.

[9] Fabio Massacci, Stephan Neuhaus, and Viet Hung Nguyen. After-life vulnerabilities: a study on firefox evolution, its vulnerabilities, and fixes. In *International Symposium on Engineering Secure Software and Systems*, pages 195–208. Springer, 2011.

[10] Nuthan Munaiah, Felivel Camilo, Wesley Wigham, Andrew Meneely, and Meiyappan Nagappan. Do bugs foreshadow vulnerabilities? an in-depth study of the chromium project. *Empirical Software Engineering*, 22(3):1305–1347, 2017.

[11] Stephan Neuhaus, Thomas Zimmermann, Christian Holler, and Andreas Zeller. Predicting vulnerable software components. In *Proceedings of the*

14th ACM conference on Computer and communications security, pages 529–540. ACM, 2007.

[12] Anh Tuan Nguyen, Michael Hilton, Mihai Codoban, Hoan Anh Nguyen, Lily Mast, Eli Rademacher, Tien N Nguyen, and Danny Dig. Api code recommendation using statistical learning from fine-grained changes. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 511–522. ACM, 2016.

[13] Viet Hung Nguyen, Stanislav Dashevskyi, and Fabio Massacci. An automatic method for assessing the versions affected by a vulnerability. *Empirical Software Engineering*, 21(6):2268–2297, 2016.

[14] Viet Hung Nguyen and Fabio Massacci. The (un) reliability of nvd vulnerable versions data: An empirical experiment on google chrome vulnerabilities. In *Proceedings of the 8th ACM SIGSAC symposium on Information, Computer and communications security*, pages 493–498. ACM, 2013.

[15] Mateusz Pawlik and Nikolaus Augsten. Tree edit distance: Robust and memory-efficient. *Information Systems*, 56:157–173, 2016.

[16] Li ping Zhang and Dong sheng Liu. Ast-based multi-language plagiarism detection method. In *2013 4th IEEE International Conference on Software Engineering and Service Science (ICSESS)*, pages 738–742. IEEE, 2013.

[17] Yonghee Shin, Andrew Meneely, Laurie Williams, and Jason A Osborne. Evaluating complexity, code churn, and developer activity metrics as indicators of software vulnerabilities. *IEEE Transactions on Software Engineering*, 37(6):772–787, 2011.

[18] Orcun Temizkan, Ram L Kumar, Sungjune Park, and Chandrasekar Subramaniam. Patch release behaviors of software vendors in response to vulnerabilities: an empirical analysis. *Journal of Management Information Systems*, 28(4):305–338, 2012.

[19] Christopher Theisen, Kim Herzig, Brendan Murphy, and Laurie Williams. Risk-based attack surface approximation: how much data is enough? In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 273–282. IEEE, 2017.

[20] Julian Thomé, Lwin Khin Shar, Domenico Bianculli, and Lionel Briand. Search-driven string constraint solving for vulnerability detection. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 198–208. IEEE, 2017.

[21] Su Zhang, Doina Caragea, and Xinming Ou. An empirical study on using the national vulnerability database to predict software vulnerabilities. In *International Conference on Database and Expert Systems Applications*, pages 217–231. Springer, 2011.