1 Introduction

Mainstream software development is increasingly relying on reuse of open source software components and libraries as well as third party APIs. Although this approach helps shorten the time to market for software vendors, it comes with security implications. Software vendors need to have an insight into the security of the components, libraries and APIs that they are importing into their products. Furthermore, they need to keep track of versions of these reused libraries so they can avoid reusing an outdated or vulnerable version. Given the pressure onto protecting users privacy as well as ensuring data integrity, confidentiality and services availability, software vendors and security researchers need to have a mechanism that facilitates the management and communication of vulnerabilities. In this scenario, organizations leverage on existing sources of vulnerabilities information (such as security bulletins, advisories and vulnerability databases) to identify potential exposures in their products, quantify potential impacts and adopt mitigations mechanisms for minimizing the consequences of the breach. Similarly, researchers rely on these sources to develop new techniques and to conduct empirical studies for advancing the field of cybersecurity.

The main problem is that organizations and researchers have to manually browse a variety of data sources to find a set of vulnerabilities of their interest and to characterize the impacts of them. Such manual process is time-consuming and error prone. Moreover, these sources of vulnerability information are often inaccurate or incomplete [4], such as the software releases that are affected by a given vulnerability [7]. Therefore, it is important to fix these inconsistencies as well as to deploy a mechanism that allows automatic reasoning of vulnerabilities and that minimizes the burden of manually exploring a large amount of security problems.

To overcome these challenges, we had proposed an automated framework for characterizing vulnerabilities. In this project, we had the following goals:

- To expand and empirically validate NIST’s standard vocabulary for describing and sharing vulnerabilities that could enable automation of security-related activities.

- To develop Natural Language Processing (NLP) techniques to automatically tag and collect vulnerability attributes from a report as well as several other sources. This toolkit will help researchers to characterize CVEs, augment NVD data, and also cross-check CVEs with other available resources such as project’s GitHub repository. This feature will significantly reduce the manual effort involved in characterizing vulnerabilities.
To develop automated techniques to search, collect and categorize available exploits for each CVE.

To automate the process of tracing vulnerabilities to software releases, components and lines of code and identify public patches for that component.

To automatically establish vulnerability life-cycle for each CVE, which contains time of introduction, time of report and time of fix.

Collect the patches and files affected for CVEs in source code repositories;

In the remaining sections of this document, we report the methodology followed to achieve each of these goals as well as their results.

2 Expanding and Empirically Validating NIST’s Standard Vocabulary Information

2.1 Current State of Research and Related Work

One crucial aspect to enable automation of vulnerability-related activities, is to establish a common vocabulary for describing vulnerabilities. This vocabulary needs to capture all the inherently characteristics and properties of vulnerabilities. With this respect, the National Institute of Standards and Technology released a draft of the Vulnerability Description Ontology (VDO) [6]. Figure 1 shows a diagram of the entities discussed in the NIST’s ontology. As shown in this figure, the VDO enumerates the concepts that are needed for accurately describing a vulnerability, such as Vulnerability, the source of information (Provenance), affected Products, and so on. These entities can be either mandatory, recommended or optional.

Therefore, as part of this work, we expanded and are currently empirically validating the NIST’s standard vocabulary (VDO), that could enable automation of security-related activities.

2.2 Results

In the next sections we discuss our work on expanding (Section 2.2.1) and validating (Section 2.2.2) the ontology.

2.2.1 Expanding the NIST’s Vulnerability Description Ontology

Through systematically looking at sources of vulnerability information (such as issue tracking systems, security advisories, mailing lists, etc), we observed that the VDO does not have entities to capture certain characteristics of vulnerabilities. Specifically, we refined the ontology regarding the following aspects (as shown in Figure 2):

- **Affected Software Components**: Currently the VDO does not identify which part of a software system are affected by a given vulnerability. Thus, we propose to add an entity (Affected Component) to the ontology that captures this type of information and indicates which part of the Product was affected. The information captured by this entity can be either at a fine-grained level (such as the vulnerable source code) or at a more coarse-grained level (e.g. software modules or components).
Figure 1: Diagram of the NIST’s Vulnerability Description Ontology. For the sake of clarity, we have hidden some elements (i.e., those that are just a refinement of attributes)

Figure 2: Refinements for the Vulnerability Description Ontology

- **Attack Patterns**: The VDO captures in the Impact Method element a description of the method used by an intruder to exploit the vulnerability. However, it currently does not fully enumerate all possible methods used by attackers to exploit vulnerabilities. Therefore, we leveraged an existing dictionary of known attack patterns, the Common Attack Pattern Enumeration and Classification (CAPEC) \(^1\), to identify common methods utilized to exploit the vulnerability and refine the values from this Impact Method entity. This way, we link a vulnerability to an entry of the CAPEC to better describe the type of attack that can be used to exploit the problem.

- **Severity**: The VDO currently does not explicitly indicate the severity of the vulnerability. Thus, we added a new entity to the ontology to document the severity of the breach. This Severity entity captures: (a) the severity level and (b) the source for this information. For

\(^1\)http://capec.mitre.org/
instance, the CVE-2014-8642 is marked as “low” severity in the vendor’s security advisory\(^2\). In this scenario, the added Severity entity would have its value attribute tagged as “low” whereas the source attribute would be equals to the URL of the advisory. Similarly, we can also compute the severity for the vulnerability using the CVSS version 2, and tag the vulnerability with the correspond computed severity value and the CVSS v2 as the source. This could enable a detailed observation of the severity according to multiple points of view.

- **Exploitation**: Once a new report arrives to a software vendor, the reporter may provide a proof of concept. This proof of concept consists of a code snippet that can be used to exploit the vulnerability. These code snippets are referred as exploitation code. While the Impact Method entity in the VDO identify the type of method used in to exploit the breach (e.g. “Authentication Bypass”), we also propose on providing concrete code snippet examples on how these breaches can be exploited. Thus, we add a new entity (Exploit Code) to capture these code snippets.

- **Evidence of Information**: The VDO currently documents the source of information of a Vulnerability entity through the Provenance element. Since we aim to collect data from many sources, we need to provide evidence for each collected information for an element in the ontology. Therefore, instead of having the Provenance entity connected only to the Vulnerability entity, it is connected to all other entities of the ontology. This way, we can observe from which source each data attribute came from.

- **Vulnerability Lifecycle**: Each vulnerability typically follows the timeline shown in Figure 3. From this figure we note that once a vulnerability is firstly introduced in the software, it can be later discovered by users, researchers and/or the development team. After this discovery, these issues are typically reported to the software vendor and are then added to the project’s issue tracking system with an open status. These entries in the issue tracking system are commonly kept private to avoid attackers exploiting them. Then, the software’s development team would discuss possible ways to mitigate the issue and then develop and release a patch that contains the source files that were modified and/or added to fix the problem. On the occasion of the release of the patch, the issue is marked as closed in the issue tracking system and the entry is disclosed to the public. Thus, we refined the ontology to add an entity for capturing the vulnerability life cycle that contains the time of introduction, report and fix. Keeping track of the dates a vulnerability was reported and fixed, allows organizations to get insights into the time that is typically needed to fix certain types of issues.

2.2.2 Empirically Validating the NIST’s VDO

We are also empirically validating it with real systems. The goal of this validation is to verify to which extent the VDO can accurately describe vulnerabilities. To do so, we are currently performing the following two major steps. In the first step, we are selecting software products across different domains. These case studies are being selected according to the following criteria:

- It is open source;
- It has a high number of disclosed vulnerabilities;
- Its vendor releases security advisories over the time;
- They have a public issue tracking system;

Once these case studies are selected, the second step is to combine the information about a vulnerability across many sources (its issue tracking system, advisories, etc) and describe them using the ontology. Through this activity, we will be able to observe how accurately the ontology was able to capture all the various information from vulnerabilities. Moreover, it also helps to verify whether there is any sort of information that was not fully captured by the ontology. This could help to further adjust the ontology (if needed).

3 An Automated Approach for Characterizing Vulnerabilities

3.1 Current State of Research and Related Work

Vulnerabilities are often publicly disclosed in many sources, such as mailing lists, the software vendor’s Website (security advisories), security bulletins, issue tracking systems as well as vulnerabilities databases. These data sources are populated on a report-basis approach, in which individuals (such as internal developers, security researchers, and end-users) communicate the found vulnerabilities by describing it in a free text format. Given the increasing number of reports that analysts have to review, such manual approach leads to problems related to incomplete information and scattered information that makes it difficult for organizations and researchers to conduct their activities, as these data sources may even have conflicting information about the same vulnerability problem. Previous research proposed the use of techniques for automatically extracting a limited set of security entities from free-text [3]. However, we currently lack an automated end-to-end approach that can automatically characterize vulnerabilities, thereby reducing the efforts related to management of vulnerabilities.

3.2 Results

To minimize the efforts on manually characterizing vulnerabilities, we developed an automated support to populate the vulnerability ontology. In short, we automatically collect information about vulnerabilities from many data sources and format the collected information according to the ontology. To achieve this goal, we applied information retrieval techniques and machine learning techniques to collect information. Figure 4 shows the overall process for developing this automated solution. Below we concisely describe each step performed in this automated approach.

**Information Retrieval Technique**: As shown in Figure 4, we applied data mining techniques to retrieve information about vulnerabilities across many sources. We gathered information from
vulnerability reports, security advisories, issue tracking systems, vulnerability databases, mailing lists and exploitation code databases. At the end of this step, we will have a unstructured dataset of vulnerability information.

**Supervised Learning Technique:** While some sources, may provide vulnerability information in a machine-readable format (e.g. XML), others sources provide vulnerability information in a free-text format. Each of these sources may contains different types of information about a vulnerability (i.e., may provide only information to populate a subset of entities from the ontology). Thus, we applied several classifiers for recognizing entities of the ontology from free-text. To do so, we firstly established a **golden standard** which corresponds to a manually established dataset of annotated vulnerability information. This dataset is later used to train our classifier algorithms to recognize entities from free text. For this technique we attempted several different classification algorithms to determine the characteristics of a CVE from its description. To achieve this, a training dataset was created consisting of CVE descriptions and manually-labeled entities for the CVE. Then, we trained eight different classification techniques with the training data using 10-fold cross-validation over 394 CVE descriptions to verify which was able to correctly classify the most descriptions. Table 1 indicates the performance of each classification algorithm. We found that **SVM** and **Naive Bayes** classification worked the best for the given data.

<table>
<thead>
<tr>
<th>Classifier</th>
<th># Correctly Classified</th>
<th># Incorrectly Classified</th>
<th>Kappa Statistic</th>
<th>Mean Absolute Error</th>
<th>Root MSE</th>
<th>Relative Absolute Error</th>
<th>Root Relative SE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Naive Bayes</td>
<td>280</td>
<td>113</td>
<td>0.6944</td>
<td>2.77%</td>
<td>15.97%</td>
<td>30.87%</td>
<td>75.44%</td>
</tr>
<tr>
<td>Decision Table</td>
<td>185</td>
<td>208</td>
<td>0.4298</td>
<td>8.10%</td>
<td>19.44%</td>
<td>90.35%</td>
<td>91.79%</td>
</tr>
<tr>
<td>Kstar</td>
<td>20</td>
<td>373</td>
<td>0.0549</td>
<td>9.07%</td>
<td>21.30%</td>
<td>101.11%</td>
<td>100.57%</td>
</tr>
<tr>
<td>SVM</td>
<td>280</td>
<td>113</td>
<td>0.6934</td>
<td>8.67%</td>
<td>20.54%</td>
<td>96.67%</td>
<td>97.00%</td>
</tr>
<tr>
<td>BayesNet</td>
<td>222</td>
<td>171</td>
<td>0.5363</td>
<td>10.71%</td>
<td>61.22%</td>
<td>78.80%</td>
<td>78.80%</td>
</tr>
<tr>
<td>Random Forest</td>
<td>259</td>
<td>134</td>
<td>0.6351</td>
<td>0.0654</td>
<td>0.1706</td>
<td>72.93%</td>
<td>80.57%</td>
</tr>
<tr>
<td>Random Tree</td>
<td>153</td>
<td>240</td>
<td>0.3507</td>
<td>0.583</td>
<td>2399</td>
<td>64.97%</td>
<td>113.30%</td>
</tr>
<tr>
<td>One R</td>
<td>65</td>
<td>329</td>
<td>0.0504</td>
<td>7.97%</td>
<td>28.24%</td>
<td>88.88%</td>
<td>133.35%</td>
</tr>
</tbody>
</table>

4 Search and Collect Available Exploits for Each CVE

4.1 Current State of Research and Related Work

For an organization whose code has been affected with vulnerabilities, it is important to know the exploitation code that exposes the consequence of the issue as well as provide an indication of the vulnerable part of their code that can be exploited. By knowing this, they can be better prepared.
to counteract potential attacks. Currently, a way for software vendors to obtain exploitation code is to ask reporters to provide these exploit codes, commonly referred as proof of concept, once they submit a new vulnerability report. Alternatively, they can also search for such code snippets in databases, such as the Exploit Database Web site \(^3\) or looking at mailing lists \(^4\). Even though the information obtained from these sources is valuable and valid, it is not organized in a standardized way that could be easily searchable. To address this problem, we propose a solution that would gather scattered information about exploits and standardize that information for users.

### 4.2 Results

We implemented a solution that searches, extracts and categorizes exploit codes. This solution is implemented in two major steps (Figure 5). In the first step we obtained information on exploits from the *Exploit Database, Security Focus* and *Issue Tracking Systems*. In the second step, we extract only the relevant code snippets from the raw data extracted from these three data sources.

The structure of the presentation of the information is similar across the three sources. For instance, in Figure 6, we can observe an exploitation code provided as part of a security report in an issue tracking system.

![Figure 5: Step by step search, retrieval and categorization of exploits for CVE instances](image)

In the Exploit Database, the information of importance, namely the code snippets that can be used to exploit vulnerabilities, are contained in the *Proof of Concept* field. They usually also have a CVE ID tagged to refer to the CVE which that code snippet would exploit. Similarly, the *Security Focus* Web site \(^5\) also may provide the exploit code for a given CVE. The information about exploitations could be obtained from issue tracking systems. Similarly to Exploit Database and mailing lists, issue tracking systems have information on the CVE ID and the specific code snippets that can be used to exploit it, among other information.

Therefore, in the first step, we made use of Web mining and information retrieval techniques in order to obtain data from the *ExploitDB, Security Focus* and *Issue Tracking Systems*. In this approach, we developed custom Web crawlers that parse these Web sites and leverage CVE IDs on the Web pages to trace the CVE instance to the exploit code.

Once the information has been gathered, it needs to be processed (Step 2). In each of these reports, there is information which is irrelevant to our goal. As shown in Figure 6 some content of the Web page is not an exploit code, but purely metadata or descriptions. Therefore, to parse and retrieve only the relevant information, i.e. the code snippets and the CVE ID, we applied

---

\(^3\)https://www.exploit-db.com/

\(^4\)http://seclists.org/

\(^5\)http://www.securityfocus.com/
heuristics. These set of heuristics attempt to detect chunks (blocks of text) that contain source code which corresponds to actual exploit codes.

At the end of this second step, this program organizes all the retrieved exploit codes per CVE instance (as depicted in the last step of Figure 5).

5 Tracing Vulnerabilities to Software Releases

5.1 Current State of Research and Related Work

The presence of vulnerabilities in a software project raises concerns about the security of the earlier versions of the project. In order to identify which versions are vulnerable, it is necessary to identify the lines of code, the component or the file that contributed to making that version vulnerable. This is usually done once the vulnerability has been fixed. The fixed version is compared to the previous version, identified as vulnerable, to produce the vulnerable lines of code. The artifact that contains the vulnerability serves as basis to compare the other versions to establish whether they are vulnerable or not. A widely currently used practice to identify earlier versions of a project touched by a vulnerability relies on manually inspecting source code files. This practice imposes two main problems. First, relying on manual inspection can be unreliable and inconsistent, which may lead to differing conclusions. Second, the process may turn out to be time consuming and inefficient.

There have been research attempts in this area to redirect the vulnerability detection in earlier versions towards automation [1,5]. The problem with the methods proposed is that they involve
a threshold level that files have to pass in order to be considered vulnerable [1]. The thresholds proposed in most of the cases come from the authors of those methods, which may reflect their biases. In other cases, the authors consider an entire file or component marked as vulnerable in the instances in which the vulnerability was fixed through adding lines of code; moreover, if a version contains at least one line of code of the vulnerability footprint, the authors agree that the entire version should be considered as vulnerable [5]. This may lead to a higher number of false positives.

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. We developed a tool (named as Patchilyzer) that traces these vulnerabilities to multiple software releases. We also evaluated the tool through using real vulnerabilities from the Apache Tomcat project. In the subsections that follow we explain our approach as well as the evaluation results.

5.2 Results

Our approach works by representing the patch in an AST (Abstract Syntax Tree) format, and then looking 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 can 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 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 ADO between the vulnerable version $V$ and a subject version $X$ and the PAR. Considering this and our goal to categorize the versions in three categories, our work is based on the following assertions:

---

$^6$ADO stands for **AST Difference Output**. It represents the differencing output between any two versions of the same product in an AST format.

$^7$PAR stands for **Patch AST Representation**. It represents the ADO between the vulnerable version $V$ and fixed version $F$. 

---
• 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.

![Figure 7: Summary of the Approach](image)

An overview of our approach is provided in Figure 7. In the following sections, the steps of our approach are described in detail.

5.2.1 AST differencing: PAR and ADO

The first step of our approach is performing the abstract syntax tree differencing of the vulnerable, subject and fixed version. We use Gumtree [2] to generate the abstract syntax tree changes between any two versions. Gumtree takes as input two versions of a software product (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, parent’s value (or label), and parent’s ID as well, and the index in the parent’s node children array.

Using Gumtree we generate the PAR, which is the set of all changes from the vulnerable version to the fixed version presented as changes in AST nodes. 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.

5.2.2 Origin Nodes Identification

Subsequently, we perform the identification of origin nodes. Origin nodes are those nodes that introduced the vulnerability in the first place. To identify these nodes, we leveraged the information in the PAR and filtered it with several heuristics. The set of origin nodes is crucial 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.

5.2.3 Similarity Check

After the PAR, ADO and Origin Nodes 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, then 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 against a user-defined threshold.

5.2.4 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 [2]. 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.

5.2.5 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.

5.2.6 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.

5.2.7 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 aimed to verify if there was any correlation between more information regarding origin nodes and higher accuracy. We considered the following features: Node Type, Node Label, Parent Node Type, Parent Node Label, and Children as valuable points of information. The five different versions of Patchilyzer (and the features each of which use) are presented in Table 2.

5.3 Evaluation
To evaluate our technique, we first retrieved 39 CVEs and 174 releases of Apache Tomcat. However, the testing 39 CVEs across 174 results in 6786 different possible combinations. Therefore, we randomly chose 35 combinations of CVE and software release to evaluate Patchilyzer. For each CVE in the randomly chosen set, we retrieved the paths to the vulnerable and fixed versions, 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. The experiments were carried out across the

<table>
<thead>
<tr>
<th>V.</th>
<th>Definition</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>NT + PT + Location</td>
<td>TypeDeclaration 32 190</td>
</tr>
<tr>
<td>II</td>
<td>NT + Children</td>
<td>TypeDeclaration [43@@Map]</td>
</tr>
<tr>
<td>III</td>
<td>NT + NL + Children</td>
<td>TypeDeclaration int [43@@Map]</td>
</tr>
<tr>
<td>IV</td>
<td>NT + NL + PT + Children</td>
<td>TypeDeclaration int 32 [43@@Map]</td>
</tr>
<tr>
<td>V</td>
<td>NT + NL + PT + PL + Children</td>
<td>TypeDeclaration int 32 Block [43@@Map]</td>
</tr>
</tbody>
</table>

Table 2: The Five Versions of the Patchilyzer
Table 3: Accuracy Results for Patchilyzer’s Versions and Thresholds

<table>
<thead>
<tr>
<th>V./ Thresholds</th>
<th>0.7</th>
<th>0.8</th>
<th>0.9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Patchilyzer I</td>
<td>0.8</td>
<td>0.77</td>
<td>0.66</td>
</tr>
<tr>
<td>Patchilyzer II</td>
<td>0.83</td>
<td>0.74</td>
<td>0.57</td>
</tr>
<tr>
<td>Patchilyzer III</td>
<td>0.86</td>
<td>0.8</td>
<td>0.69</td>
</tr>
<tr>
<td>Patchilyzer IV</td>
<td>0.83</td>
<td>0.74</td>
<td>0.63</td>
</tr>
<tr>
<td>Patchilyzer V</td>
<td>0.86</td>
<td>0.78</td>
<td>0.66</td>
</tr>
</tbody>
</table>

The collected data was manually evaluated by four reviewers with relevant security 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). 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.

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 3. As shown in this table, Patchilyzer III demonstrates higher accuracy than all other versions, 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 Patchilyzer 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.

6 Collect Affected Files for CVEs in Source Code Repositories

In this project we developed a program that extracts fix patches (i.e., the set of modified files within a commit that fixed a CVE) and the “unidiff” of the affected files (i.e., the source code differencing of the commit in the unidiff format).

6.1 Results

To achieve this goal, we developed a program shown in Figure 8. This program has three inputs:

- a path to a CSV file that contains a list of GitHub repositories;
- a path to where the project’s repository will be cloned;
- a path indicating the destination folder in which all the data will be placed;

Figure 9 depicts the steps for extracting patches from source code repository. First, it retrieves a list of all commits from the project’s repository. Subsequently, it verifies the commits that fix a
CVE through performing a string matching (i.e., CVE-[0-9]+[0-9]+). After matching the CVE-related commits, it extracts the patch (containing all the affected files) along with their unified differences (“unidiff”). The data extracted by this program is saved locally in a folder, whose location is specified by the user.

7 Vulnerability Life Cycle Management

Once a project is vulnerable, organizations need to keep track of the status of the vulnerability. In other words, it is important to verify whether the fixing patch was already released, so organizations can apply the fix to their software products that use the vulnerable code. Keeping track of the dates a vulnerability was reported and fixed, allows organizations to get insights about the time it typically takes to fix certain types of issues. Thus, we are developing a technique to facilitate the management of vulnerability life cycle by keeping track of the status of the vulnerability over time. The solution gathers, processes and standardizes information on the vulnerabilities’ lifecycle (i.e. time of introduction, time of report and time of fix), across different software projects.

As of now, we are able to successfully identify the time of introduction and fix of vulnerabilities. We are currently finalizing our technique for detecting the time of report.

7.1 Results

The process to capture the lifecycle of vulnerability occurs as follows:

- **Time of introduction identification**: we leveraged our vulnerable release identification technique (described in Section 5) to identify what was the first vulnerable release. Through observing the commit that first had the vulnerability, we infer the time the vulnerability was introduced in the code.
• **Time of report identification:** the time of the report is identified through two different ways: (i) via issue tracking systems and (ii) full disclosures. The first approach involves mining the issue tracking system of the corresponding vulnerability’s software vendor to identify when the issue was reported to the vendor (i.e., when a bug was filled for the vulnerability). The second approach relies on vulnerability timeline information available in full disclosures.

• **Time of fix identification:** we leverage our techniques that extract patches from source code repositories (Section 6). For that, we extract the date of the commit that fixed the vulnerability (CVE instance). The date of the commit corresponds to the time of fix for the CVE.

**References Cited**


