POLITECNICO DI TORINO

Corso di Laurea in Computer Engineering

Tesi di Laurea Magistrale

Discover Vulnerabilities in Open-Source through Mining Software Repositories

Relatore
prof. Paolo Camurati

Candidato
Giuseppe Pacelli

Aprile 2017

Abstract

Because of the many advantages offered by open source software, the number of companies using it has increased over the years and today many commercial products include open source software. Unfortunately, every application that includes an open source library affected by a certain vulnerability is potentially affected by the vulnerability as well.

As part of a secure software development lifecycle, application vendors need to determine on a regular basis if any of the used open-source software libraries contains a vulnerability. Today, those checks commonly rely on the comparison of the identifiers of the library in use with those mentioned by so-called vulnerability databases. This process largely depends on natural-language vulnerability descriptions and expert knowledge, which makes it time-consuming and error-prone.

The source code of open-source software is typically managed by publicly accessible versioning control systems. This makes it possible to analyze the source code changes introduced by open-source developers in order to fix a given vulnerability. Such changes to the source code are submitted to the versioning control system by means of commits and can comprise the addition, deletion or modification of one or more method bodies. The commits that fix a given vulnerability are often mentioned in bug tracking systems and security advisories.

In this thesis I will propose two different approaches to automatically assess whether or not a given library is vulnerable with respect to a given bug. The common basis for both approaches is the abstract syntax tree representation of the source code of both vulnerable and fixed method bodies as well as the corresponding change edit script, i.e., the list of edit operations required to transform the syntax tree of the vulnerable revision into the syntax tree of the fixed revision.

As part of the first approach, a given library in use is decompiled and the resulting source code is analyzed in order to see whether the source code changes introduced by an open-source developer are present in the library at hand or not.

The second approach does not directly analyze a library used by a given application, but analyses a bulk of library versions downloadable from software repositories such as the Maven Central. Very often, both the compiled code as well as the corresponding source code can be downloaded from such repositories, and the approach searches for the presence of vulnerable or fixed revisions in order to establish whether a given version is subject to a vulnerability or not. If method bodies are not equal to either the vulnerable or fixed revision, the approach computes the distance to both of the revisions in order to establish whether a method at hand is closer to the vulnerable or the fixed revision.

In this thesis I will explain the main limitations of the first approach that lead to the development of the second one, in order to be more effective and reliable in the performed assessments and, at the same time, assess a larger number of libraries, as reported with the obtained results.

Contents

Chapter 1

Introduction

1.1 Motivation

Open source software is software with source code that everybody can potentially read, modify and execute. Because of these features, it has a lot of advantages over commercial closed source software, worth mentioning are:

- **Security**: Because of its availability to a very large audience, all bugs and vulnerabilities contained in open source software are discovered and fixed very quickly, compared to closed source software where only a limited set of people has the possibility to check for the flaws.
- **Quality**: Open source software can be customized and reviewed by large communities of developers, therefore getting closer to what user really need/want.
- **Cost**: Not to mention the cost of the purchase of the software itself, the cost of maintaining and updating software is really high. Open source software offers everything for free.

Because of these advantages, the number of companies using open source is increasing over the years and today many commercial products include open source libraries. If we only consider the years between 2012 and 2014, the number of downloads of java artifacts from Maven central has doubled from 6B to 13B yearly downloads[1]. There is a drawback: when a vulnerability concerning an open source library is disclosed, all the products including that library are potentially affected by the vulnerability. A clear example of this issue can be found in the well known Heartbleed[2] bug, when the security of all the applications including OpenSSL[3] library, affected by the bug, was severely compromised. OWASP[4] acknowledged the criticality of this issue by including it in the OWASP list of the Top-10 security vulnerabilities for 2013 as "A9-Using components with known vulnerabilities"[5]. Each element of the top 10 is evaluated for security weakness by prevalence and detectability, in this case they are, respectively, widespread and difficult. The obvious solution to this problem is to update to a more recent and patched library but, unfortunately, very often the vulnerable libraries are found to be still in use quite some time after the relative patch has been released. A possible explanation for this is that, if the disclosure and relative fix happens at development time it is rather easy just to include the patched version of the library, whereas it becomes more cumbersome when the system has already been deployed and made available to the final users. The key question that vendors have to answer is whether or not a given vulnerability, that was found in an OSS library used in one of their products, is indeed exploitable given the particular use that such a product makes of that library. Nowadays the answer to this question is mostly a manual process, therefore time-consuming and error-prone. Coming to a right conclusion with the manual approach is very difficult, since it requires a deep knowledge of the use that application makes of the vulnerable library.

OWASP tries to give an automated solution to the problem with its Dependency Check tool[6]. When vulnerabilities are disclosed, they are shortly described in high-level textual descriptions

written in natural language, and in the case of NVD[1] they are associated with a CVE[2] and a CPE[3] number. OWASP Dependency Check scans the project to be analyzed and tries to assign a CPE to the libraries found. Once all the possible CPEs have been assigned, they are checked against the associated CVEs. This approach would be always working if for every CVE there was always the complete list of all the affected CPEs, which does not happen because often only major releases are listed, not to mention transitive dependencies. Another limitation for Dependency check is the way it maps libraries to CPEs, which is done by strings and artifact found within the app, but this is highly subject to both false positives and negatives. Last but not least, Dependency check does not give any information about the exploitability of the vulnerability within a given context, just like most of the existing commercial solutions that all share a similar approach.

It is very important to find an automatic approach that allows for determining if the libraries in use by the application are vulnerable or fixed, so that both the uncertainty caused by the manual assessment and the time needed to give such assessment are not a problem anymore.

1.2 Existing solution

In the above mentioned context vulas[7] (Vulnerability Assessment) is a tool based on an approach that looks at the source code in order to determine whether or not vulnerabilities affecting open source libraries are relevant (exploitable) in a given application context.
The approach is based on the following assumption: whenever an application that includes a vulnerable library executes a fragment of code in the library that would be updated in a security patch, there is a significant risk that the vulnerability can be exploited. The basic idea is to identify the library's methods/constructors that have been modified in order to fix the vulnerability, and check if they are actually executed whenever the application makes use of the library.

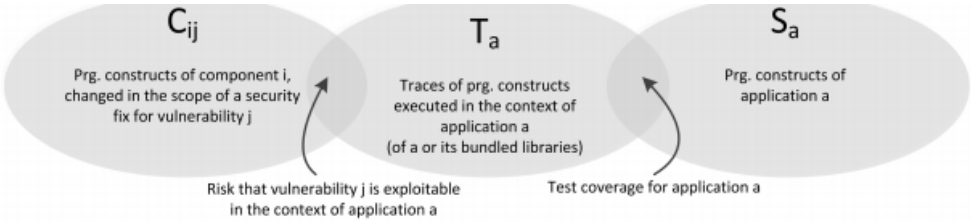


Figure 1.1: Change list, traces and programming constructs

In order to perform the assessment, the so called change list is used. The change list is the list of methods and constructors changed as a part of the fix for a given security vulnerability. The change list is built by comparing the code of the released patch to the original (and vulnerable) code, starting from the commits found in the version control system where the library is developed.

Three different assessment levels allow vulas to decide whether or not a vulnerability is exploitable:

1. The application bundles an OSS component that includes an element of the change list.

   This can be determined by intersecting the elements of the change list with the set of all methods and constructors of all dependencies bundled within a given application.

[1] U.S. government repository of standards based vulnerability management data represented using the Security Content Automation Protocol (SCAP)

[2] Common Vulnerabilities and Exposures

[3] Common Platform Enumeration. It is a structured naming scheme for information technology systems, software, and packages.

2. The application can potentially execute elements belonging to the change list.

   The word potentially is used since this assessment is done by performing static reachability analysis of the bytecode of the application and its dependencies. If the analysis concludes that a change list element is reachable, it means that the application control flow may reach the critical coding of the change list.

3. The application actually executes elements of the change list.

   This is determined by intersecting the elements of the change list and application traces collected during the actual execution (including traces of all libraries used by the application).

The intersections between change list elements, traces collected during app execution and programming constructs for the applications are shown in Figure 1.1.

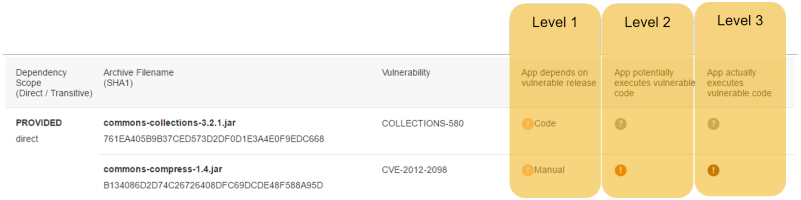


Figure 1.2: Different Assessment levels for a test application.

If there was a link between source code repositories and software repositories (that is, between repositories where the development of the library is done and repository where the packaged library is published) it would be possible to automatically assess the vulnerability status of a library as soon as the commit fixing the vulnerability was released. Unfortunately this link does not exist, therefore it is necessary to study a different approach that allows to establish the vulnerability state of the archive to be analyzed.

In order to solve such a problem, the approach proposed in this thesis is based on source code analysis. By analyzing the source code of the elements belonging to the change list of a certain bug vulas is able to verify if the packaged library actually contains the code for the fix and, therefore, assess its vulnerability state (either fixed or vulnerable). Starting from the source code of the potentially vulnerable library also makes the tool immune from repackaging of the library, transitive dependency, misconfigured pom file and makes that the text description of the vulnerabilities (as provided by NVD) is not needed anymore.

The basic approach consists in :

1. Discovering which libraries included in the application under analysis are affected by a certain bug;
2. Retrieving the change list for that bug and library affected;
3. Decompiling the code of the used library and compare it with the fixed and vulnerable code for that pair bug and library;
4. According to the comparison, decide if the library in use is vulnerable or fixed.

1.2.1 Architecture and Implementation

Within vulas, every application can be analyzed through a command line interface (vulas-cli) or a maven plugin (vulas-mvn-plugin). All information collected during the various analysis is stored in a central database, and consumed by security experts of the respective application with help of the frontend uis (vulas-app-ui and vulas-bug-ui). The main purpose of the server (vulas-server) is to store and retrieve all the information from the database and use them to provide the user with a report on the vulnerabilities present in its software. A general overview of the architecture is shown in Figure 1.3.

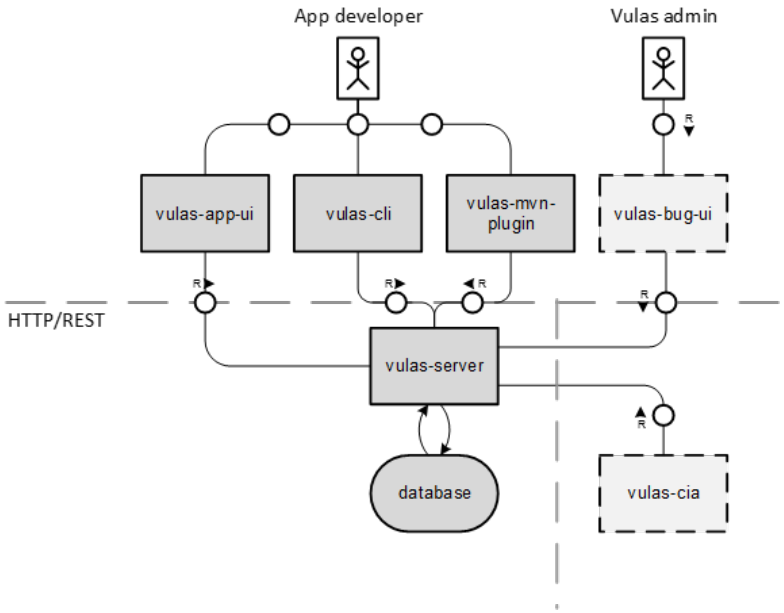


Figure 1.3: Vulas Architecture

The implementation of vulas has been designed to support the vulnerability assessment process into open source library included into Java software and, for this reason, it can be easily integrated into a typical Java development environment. This has been achieved using Apache Maven, in fact vulas can be included into any java application just adding its maven dependency into the maven file of the host application. Then every tool and operation that can be done by the tool is executed using a different maven custom goal.
The current implementation for the various components is:

- **Database**
  It is based on Spring and can only be accessed by the server.

- **Vulas Server**
  Running on Apache Tomcat, it accesses the database when it is needed by another vulas component and gives back the required information through a set of RESTful APIs.

- **Maven Plugins**
  Different maven plugins are designed in order to perform the different assessment levels:

  – **Source Code Analyzer** This tool is implemented using a maven plugin and it uses a parser for the analysis of every Java class. The result of this tool is the list of signatures of every programming construct present into the application and in all the library included into it. It takes care of analyzing each library and upload the results on the server.

  – **Runtime Tracer** This tool's aim is to gather information at run-time. The easiest way to execute a large portion of an application is to run the unit tests that are supposed to check that every feature is working properly. During those executions same data about it needs to be gathered: in particular the signature of every method that is actually called in the target application and in each library connected to it is needed. In order to achieve this goal vulas is instrumenting the code: this means that vulas is injecting some bytecode into the existing bytecode of the compiled classes. Vulas injects code that actually collects and sends to the server the stack traces into each method of each class presented into the application. This means that if a method is executed this will first collect its stack trace and send it to the server and then just proceed with its own code. The tracing operation is performed either on the fly (using instrumentation) or statically both for the application and all its dependencies.

  - **Static Analyzer** The static analysis is done using two third party analysis frameworks tools, Wala[9] and Soot[10], that build a call graph of the possible executions using as starting point for the algorithm the application methods. So this tool is used to statically analyze the libraries reachability starting from the application.
  - **Version Check Plugin** This tool is implemented using a maven plugin, and it is the part of vulas where I have focused my work. The plugin aims at discovering whether or not the code of the potentially vulnerable library found in the application under analysis is the vulnerable or the fixed one.

- **App ui**
  It is the main frontend for the tool, developed using SAPUi5 framework. Its purpose it show in an easy manner all the information about the vulnerabilities affecting a certain application.

- **Bug ui**
  It is the second frontend for the tool, developed using SAPUi5 framework. Its purpose is that of making easier the understanding of the results given by the version check plugin. It is needed since it might not be immediate, for a developer approaching the tool for the first times, to understand why a certain application has been flagged as vulnerable (or fixed) basing on the command line output provided by the version check plugin. It shows all the bugs known to the server, for each of them the list of archives potentially affected and information about the version check results if the archive has been analyzed. It also allows for manually assessing a vulnerability.

  - **Patch Analyzer** Its purpose is to interact with various Version Control System. In vulas it can interact with Git and Subversion using relatively the JGit and SVNKit as external libraries, which allow vulas to retrieve the code for the fixed and for the vulnerable revision. After having gathered the two versions the tool can proceed to build the parse tree of every Java file, obtaining the change list by comparing the two versions available for the same file. The change list is then transmitted to the central vulas backend, which stores it in a central database.

- **Cia**
  The acronym stands for change impact analysis, it is a server running on Apache Tomcat. It interacts with *vulas* server and exposes some additional APIs.

1.2.2 Limitations

The normal life cycle for the patch of a security bug consists of:

1. Vulnerability is found in the library;
2. Code fixing the vulnerability is committed to source code repository;
3. Packaged version of the library including the fixed code is uploaded to software repository.

After the fix is released, nothing prevents developer from modifying again (and again) the source code of what was initially the fix for a certain bug, still keeping the library not vulnerable to that bug. Since version check approach is based on finding the source code that is part of the security fix, what if this source code is completely modified/moved? Version check will no longer be able to identify the fix and will mark, wrongly, the version as vulnerable.

A problem that is really difficult to solve is the one concerning the decompilation phase, since it is not always possible to go back to the original source code given a compiled archive. It might not only be due to the decompiler limitations, but also compilers might introduce software changes in order to improve performances or to save space.

1.3 Contributions

In the following chapters I will explain the details about how the tool work, while putting special focus on how I investigated, and overcome, the limitations. I mainly focused on two problems:

1. Existing algorithm improvements:
    - First of all, I had to investigate on various CVEs in order to understand the time-being limitations of the plugin and why the assessment was wrongly made. After this phase of analysis I was able to find out the most common problems, such as the ones linked to decompilation, and studied solutions to overcome them (when possible);
    - I will evaluate the correctness of the assessments given by the existing tool and describe the study I made in order to understand if these assessments are trustworthy or if they still need manual review;
    - In Appendix C I will describe the frontend that I developed in order to have an easy understanding of the assessments performed by the version check plugin.
2. New algorithm development:
    - Since the results given by the previous algorithm were not always reliable, I will describe the new algorithm I developed in order to automatically assess a large number of archives and trust such assessments.

Chapter 2

Signature Analysis

2.1 Description

The term signature analysis is used to indicate the process of analyzing the source code of methods and constructs bodies. Within vulas this process is performed by the version check maven plugin[1] and its purpose is to assess if the libraries used by the application under analysis are vulnerable to a certain bug. In fact, the vulas approach can easily report whether the application includes libraries containing constructs whose body was changed in order to patch a vulnerability. However it is not trivial to assess whether the construct is contained in its vulnerable or fixed version. This kind of assessment is very important when working on big projects that include a large number of external libraries, each of them potentially including vulnerable software. If this assessment had to be done by hand for each of the included libraries, a human should always:

- Whenever a new library containing potentially vulnerable code is identified, check if it is used by the application;
- Whenever a new bug affecting one of these libraries is disclosed, check if the version of that library being used is fixed or vulnerable, according to the textual description given by NVD;
- If the version in use is not among the ones listed, manually analyze the code to understand whether or not the fix is contained.

The manual approach is indeed very weak, since it is time consuming, error prone and requires a qualified person, as well as relying on the textual description given by the NVD. For large companies with a lot of on-going projects, each of them including a large number of libraries, this would be very costly in terms of money and resources. The approach proposed with the signature analysis aims at automating and making this task reliable.

2.1.1 Overview

Given an application to be tested for a certain bug, version check receives the list of potentially vulnerable libraries included in the application from vulas backend. For each of the affected libraries, another call to vulas backend retrieves the list of all the programming constructs[2] that have been changed in the library in order to apply the fix.[3]

Once the list is received it is possible to start the algorithm, looking for the fixes within the library. Each element of the change list can exists in one of the following three states:

---

[1] In the following, the version check maven plugin will be referred as version check.

[2] With the term programming constructs I intend both methods and constructors

[3] In the following, this list will be referred as change list.

Programming constructs of the change list of the OSS patch
Repository:http://svn.apache.org/repos/asf/httpcomponents/httpclient
Revisions fixing the vulnerability:1411702,1411705

Change	Revision	Type	Qualified Construct Name (Path)
DEL	1411702	Method	**org.apache.http.conn.ssl.TestHostnameVerifier.testacceptableCountryWildcards()** /httpcomponents/httpclient/trunk/httpclient/src/test/java/org/apache/http/conn/ssl/TestHostnameVerifier.java
ADD	1411702	Method	**org.apache.http.conn.ssl.TestHostnameVerifier.testAcceptableCountryWildcards()** /httpcomponents/httpclient/trunk/httpclient/src/test/java/org/apache/http/conn/ssl/TestHostnameVerifier.java
MOD	1411702	Method	**org.apache.http.conn.ssl.AbstractVerifier.verify(String,String[],String[],boolean)** /httpcomponents/httpclient/trunk/httpclient/src/main/java/org/apache/http/conn/ssl/AbstractVerifier.java
MOD	1411702	Method	**org.apache.http.conn.ssl.AbstractVerifier.getCNs(X509Certificate)** /httpcomponents/httpclient/trunk/httpclient/src/main/java/org/apache/http/conn/ssl/AbstractVerifier.java
ADD	1411702	Method	**org.apache.http.conn.ssl.TestHostnameVerifier.testGetCNs()** /httpcomponents/httpclient/trunk/httpclient/src/test/java/org/apache/http/conn/ssl/TestHostnameVerifier.java
DEL	1411705	Method	**org.apache.http.conn.ssl.TestHostnameVerifier.testacceptableCountryWildcards()** /httpcomponents/httpclient/branches/4.2.x/httpclient/src/test/java/org/apache/http/conn/ssl/TestHostnameVerifier.java
ADD	1411705	Method	**org.apache.http.conn.ssl.TestHostnameVerifier.testGetCNs()** /httpcomponents/httpclient/branches/4.2.x/httpclient/src/test/java/org/apache/http/conn/ssl/TestHostnameVerifier.java
ADD	1411705	Method	**org.apache.http.conn.ssl.TestHostnameVerifier.testAcceptableCountryWildcards()** /httpcomponents/httpclient/branches/4.2.x/httpclient/src/test/java/org/apache/http/conn/ssl/TestHostnameVerifier.java
MOD	1411705	Method	**org.apache.http.conn.ssl.AbstractVerifier.verify(String,String[],String[],boolean)** /httpcomponents/httpclient/branches/4.2.x/httpclient/src/main/java/org/apache/http/conn/ssl/AbstractVerifier.java
MOD	1411705	Method	**org.apache.http.conn.ssl.AbstractVerifier.getCNs(X509Certificate)** /httpcomponents/httpclient/branches/4.2.x/httpclient/src/main/java/org/apache/http/conn/ssl/AbstractVerifier.java

Figure 2.1: Example of change list for a certain bug.

1. **ADD** The programming construct has been added in order to fix the vulnerability;
2. **DEL** The programming construct has been deleted in order to fix the vulnerability;
3. **MOD** The programming construct has been modified in order to fix the vulnerability.

For each of the previous states a different way to check if the construct within the library under analysis is fixed or vulnerable exists:

1. Case **ADD**: no source code is needed, the fix is contained if the construct is found within the library;
2. Case **DEL**: no source code is needed, the fix is contained if the construct is not found within the library;
3. Case **MOD**: it is necessary to analyze the source code of the construct, i.e., the method or constructor body. This is done by decompiling the class containing the construct and comparing the source code obtained against the source code of the fixed version of the construct as available in the library VCS[4] repository.

In the remainder of this section I will describe how the comparison is performed for modified programming constructs.

Modified programming constructs

If for both the cases 'ADD' and 'DEL' it is very easy to check the presence of the fix, the case 'MOD' has different challenges:

- Finding a good representation for the source code;
- Decompiling the class containing the construct within the packaged library archive;
- Looking for the changes introduced by the fix in the decompiled source code.

In order to easily represent and compare the source code, the chosen representation is the abstract syntax tree.[5] An AST is a Tree representation of the source code, each node of the Tree being a construct of the source code starting from which the tree has been built.[11] E.g., the AST for the source code in Listing 2.1 is shown in Figure 2.2.

[4]VCS stands for Version Control System.

[5]In the following, Abstract Syntax Tree will be referred as AST.

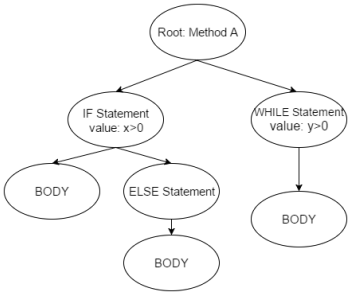


Figure 2.2: Sample AST for sample source code.

```
function A (int x, int y){
    if ( x>0 ) {
        // body
    } else {
        // body
    }

    while ( y>0 ) {
        // body
    }
}
```

Listing 2.1: Sample source code

In order to analyze the code for the construct in the library in use, it is necessary to decompile the class containing it, and this is obtained by using the open source decompiler Procyon[8]. Once the decompilation process is complete, it is possible to represent the source code of the modified construct as an AST.

The approach of the signature analysis for checking the modified constructs is based on the following assumption:

Assumption 1. *(i) If the changes necessary to transform the vulnerable version in the fixed version are the same needed for transforming the vulnerable version in the version under test, then the version under test is fixed.*
(ii) If the changes necessary to transform the vulnerable version in the fixed version are the same needed for transforming the version under test in the fixed version, then the version under test is vulnerable.

Assumption 1(*i*) is based on the fact that if a construct is fixed, no changes are required in order to fix the vulnerability. Similarly Assumption 1(*ii*) states that if a construct is vulnerable, the changes that are introduced by the patch fixing the vulnerability must still be introduced in order to fix the test version.

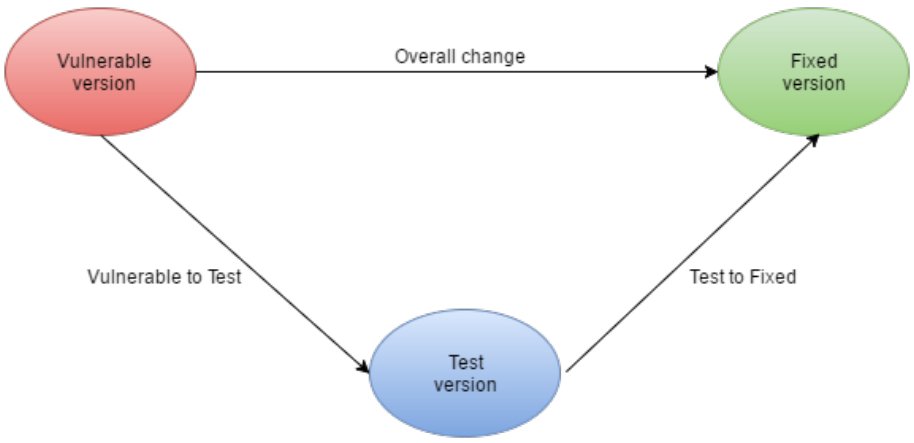


Figure 2.3: Graphical representation of Assumption 1.

The three ASTs for vulnerable, fixed and test version need to be compared to each other in order to produce the three sets of changes[6] needed for the analysis.

As shown in Figure 2.3, we have three edit scripts:

[6]In the following, the set of changes will be referred as edit script, as they are represented as the set of edit operations.

1. **Overall change** represents the changes needed in order to go from vulnerable to fixed version;

2. **Vulnerable to test** represents the changes needed in order to go from vulnerable version to version under test;

3. **Test to fixed** represents the changes needed in order to go from version under test to fixed version.

The three edit scripts are obtained by using the open source tool ChangeDistiller[12]. ChangeDistiller takes as input two ASTs and computes their difference in two phases:

1. Establish mappings (pair of nodes) between similar nodes of two ASTs;

2. Based on the computed mappings, compute the edit script (minimal set of changes that are necessary in order to go from one AST to the other one).

When a construct is modified as part of the fix, the type of changes in edit script between the two different versions can be:

- **Inserted**: node was not existing in the vulnerable version;
- **Deleted**: node has been removed in the fixed version;
- **Moved**: node has changed parent;
- **Updated**: node has changed value.

For each change it is available:

- New value of the node (not for deleted);
- Old value of the node (not for inserted);
- Parent of the node. Only for change of type modified both old and new parent are available.

Having the three edit scripts available it is possible to assess, according to the following rules, if the construct is fixed or vulnerable:

$$Fixed \iff overall\ change \cap vulnerable\ to\ test \neq \emptyset$$

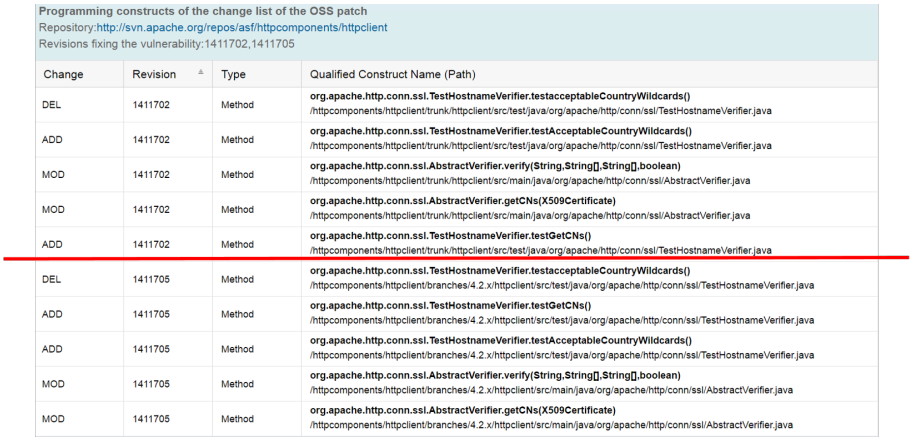
$$Vulnerable \iff overall\ change \cap vulnerable\ to\ test = \emptyset$$

Intersections are computed taking into account that type of change is the same, parent is the same and new or old values match. Although these rules would be enough to give a correct assessment, they are extended as follows in order to get a more reliable result:

$$Fixed \iff overall\ change \cap vulnerable\ to\ test \neq \emptyset \wedge overall\ change \cap test\ to\ fixed = \emptyset$$

$$Vulnerable \iff overall\ change \cap vulnerable\ to\ test = \emptyset \wedge overall\ change \cap test\ to\ fixed \neq \emptyset$$

The new rules allow for capturing both false positives (constructs that are marked as fixed, but that are actually vulnerable) and false negatives (constructs that are marked as vulnerable, but that are actually fixed).



Programming constructs of the change list of the OSS patch
Repository:http://svn.apache.org/repos/asf/httpcomponents/httpclient
Revisions fixing the vulnerability:1411702,1411705

Change	Revision	Type	Qualified Construct Name (Path)
DEL	1411702	Method	**org.apache.http.conn.ssl.TestHostnameVerifier.testacceptableCountryWildcards()** /httpcomponents/httpclient/trunk/httpclient/src/test/java/org/apache/http/conn/ssl/TestHostnameVerifier.java
ADD	1411702	Method	**org.apache.http.conn.ssl.TestHostnameVerifier.testAcceptableCountryWildcards()** /httpcomponents/httpclient/trunk/httpclient/src/test/java/org/apache/http/conn/ssl/TestHostnameVerifier.java
MOD	1411702	Method	**org.apache.http.conn.ssl.AbstractVerifier.verify(String,String[],String[],boolean)** /httpcomponents/httpclient/trunk/httpclient/src/main/java/org/apache/http/conn/ssl/AbstractVerifier.java
MOD	1411702	Method	**org.apache.http.conn.ssl.AbstractVerifier.getCNs(X509Certificate)** /httpcomponents/httpclient/trunk/httpclient/src/main/java/org/apache/http/conn/ssl/AbstractVerifier.java
ADD	1411702	Method	**org.apache.http.conn.ssl.TestHostnameVerifier.testGetCNs()** /httpcomponents/httpclient/trunk/httpclient/src/test/java/org/apache/http/conn/ssl/TestHostnameVerifier.java
DEL	1411705	Method	**org.apache.http.conn.ssl.TestHostnameVerifier.testacceptableCountryWildcards()** /httpcomponents/httpclient/branches/4.2.x/httpclient/src/test/java/org/apache/http/conn/ssl/TestHostnameVerifier.java
ADD	1411705	Method	**org.apache.http.conn.ssl.TestHostnameVerifier.testGetCNs()** /httpcomponents/httpclient/branches/4.2.x/httpclient/src/test/java/org/apache/http/conn/ssl/TestHostnameVerifier.java
ADD	1411705	Method	**org.apache.http.conn.ssl.TestHostnameVerifier.testAcceptableCountryWildcards()** /httpcomponents/httpclient/branches/4.2.x/httpclient/src/test/java/org/apache/http/conn/ssl/TestHostnameVerifier.java
MOD	1411705	Method	**org.apache.http.conn.ssl.AbstractVerifier.verify(String,String[],String[],boolean)** /httpcomponents/httpclient/branches/4.2.x/httpclient/src/main/java/org/apache/http/conn/ssl/AbstractVerifier.java
MOD	1411705	Method	**org.apache.http.conn.ssl.AbstractVerifier.getCNs(X509Certificate)** /httpcomponents/httpclient/branches/4.2.x/httpclient/src/main/java/org/apache/http/conn/ssl/AbstractVerifier.java

Figure 2.4: Change list containing two paths.

Putting everything together

Having clear how the decision is taken for a single construct, it is now possible to discuss how the assessment for the whole archive is given.

When a library is under development it often happens that more branches are developed at the same time and, when the fixed is released, it is developed on the various branches (paths).

Version check takes this situation into account in order to assess the state of the archive under test. For each of the paths all the constructs belonging to it are analyzed according to the previously described rules, and a path P is considered:

$$Fixed \iff \forall\ construct\ c \in\ path\ P \mid c\ is\ fixed$$

$$Vulnerable \Rightarrow \exists\ construct\ c \in\ path\ P \mid c\ is\ vulnerable$$

Once the assessments for all the paths has been done, the archive under test A can be considered:

$$Fixed \iff \exists\ path\ p \in\ archive\ A \mid p\ is\ fixed$$

$$Vulnerable \Rightarrow \nexists\ path\ p \in\ archive\ A \mid p\ is\ fixed$$

The pseudo code with the implementation of the algorithm for signature analysis is available in appendix A.1.

2.2 Problem analysis

In this section I will explain the limitations of the existing algorithm through a set of chosen case studies, each of them regarding a single CVE. For each of the CVEs a short description is available online and includes the group and artifact of the archive affected by the vulnerability, plus the number of the first version for which such vulnerability has been fixed. When analyzing the CVEs I particularly took care of understanding why the result given by version check for a certain version of a certain archive was different if compared with the result expected if reading the description of the fix for the CVE given by NVD.

2.2.1 Case studies

CVE-2011-1498

NVD description: *This CVE affects Apache HttpClient 4.x before 4.1.1 in Apache HttpComponents, which, when used with an authenticating proxy server, sends the Proxy-Authorization header to the*

origin server, which allows remote web servers to obtain sensitive information by logging this header.
Since version Apache HttpClient 4.1.1 is fixed, I included it as dependency in the test application and run version check, expecting to find fixed as output. This was however not the case, and version check was assessing such version as vulnerable.
We can have a look at the method *process(final HttpRequest request, final HttpContext context)*, modified as part of the fix for this bug, in Listings 2.2 and 2.3.

```
if (request.containsHeader
     (AUTH.PROXY_AUTH_RESP)) {
    return;
}

// Obtain authentication state
AuthState authState = (AuthState)
     context.getAttribute(
        ClientContext.PROXY_AUTH_STATE);
if (authState == null) {
    return;
}
```

Listing 2.2: Vulnerable version

```
if (request.containsHeader
     (AUTH.PROXY_AUTH_RESP)) {
    return;
}

HttpRoutedConnection conn =
     (HttpRoutedConnection)
     context.getAttribute(
        ExecutionContext.HTTP_CONNECTION);
HttpRoute route = conn.getRoute();
if (route.isTunnelled()) {
    return;
}

// Obtain authentication state
AuthState authState = (AuthState)
     context.getAttribute(
        ClientContext.PROXY_AUTH_STATE);
if (authState == null) {
    return;
}
```

Listing 2.3: Fixed version

As we can see from Listings 2.2 and 2.3, the fix for this vulnerability consists of:

- Declaration of variable *conn*
- Declaration of variable *route*
- Insert of the *if* statement with condition *route.isTunnelled()*
- Insert of the *return* statement within the previous *if*

What we expect from the overall change edit script is something like:

```
Insert: ;
Insert: route.isTunnelled()
Insert: HttpRoute route = conn.getRoute();
Insert: HttpRoutedConnection conn = (HttpRoutedConnection)
     context.getAttribute(ExecutionContext.HTTP_CONNECTION);
```

Instead, by debugging the code up to the point where the overall change is produced, the actual edit script is:

```
Move:
Move: (authState == null)
Update: (authScheme == null)
Insert:
Move: (authScheme == null)
Insert: (authScheme == null)
Update: (authState == null)
Insert: HttpRoute route = conn.getRoute();
```

This is bigger than the expected one, and also contains mistakes. Therefore the problem for which this bug is not properly assessed is in the creation of the overall change edit script, that doesn't allow to find the proper intersection with the vulnerable to test edit script.

```
...
}
final HttpRoutedConnection conn =
     (HttpRoutedConnection)
     context.getAttribute("http.connection");
if (conn == null) {
    this.log.debug((Object)"HTTP␣connection␣not␣set␣in␣
        the␣context");
    return;
}
final HttpRoute route = conn.getRoute();
...
```

Listing 2.4: Decompiled version

If we look at Listing 2.4, that shows the decompiled version of the construct, it is possible to notice how the decompiler:

- Adds *final* identifier to variables that were not declared as final in the original source code;
- Expands the value of the constant *ExecutionContext.HTTP_CONNECTION* with its runtime value*"http.connection"*.

Even if the overall change edit script was computed in the correct way, in this case the changes added by the decompiler would not have allowed to find this change in the code.

CVE-2014-3529

NVD description: *This CVE affects library apache-poi-ooxml. The OPC SAX setup in Apache POI before 3.10.1 allows remote attackers to read arbitrary files via an OpenXML file containing an XML external entity declaration in conjunction with an entity reference, related to an XML External Entity (XXE) issue.*
For this CVE the fix is released in the same commit in two different versions of the archive, as we can see from Apache POI website[14]:

- Version 3.11-beta1 (2014-08-04):
  56164 - Tidy up the OPC SAX setup code with a new common Helper, preventing external entity expansion (CVE-2014-3529) -r1569991
- Version 3.10.1 (2014-08-18):
  56164 - Tidy up the OPC SAX setup code with a new common Helper, preventing external entity expansion (CVE-2014-3529) -r1569991

For both the version the version check plugin was giving the wrong assessment, marking them as vulnerable. Since version Apache Apache-poi-ooxml 3.10.1 is fixed, I included it as dependency in the test application and run version check, expecting to find fixed as output. This was however not the case, and version check was assessing such version as vulnerable. We can have a look at Listings 2.5 and 2.6 for one of the construct wrongly assessed, *POIXMLDocument(OPCPackage)*. As fix for this bug, method *POIXMLDocument(OPCPackage)* has been modified.

```
protected POIXMLDocument(OPCPackage pkg) {
        super(pkg);
        this.pkg = pkg;
}
```

Listing 2.5: Vulnerable version

```
protected POIXMLDocument(OPCPackage pkg) {
        super(pkg);
        this.pkg = pkg;
        SystemCache.get().setSaxLoader(null);
}
```

Listing 2.6: Fixed version

The overall change for this construct is:

```
Insert: SystemCache.get().setSaxLoader(null);
```

Despite it is a single change, it is not found.

```
protected POIXMLDocument(OPCPackage pkg) {
        super(pkg);
        this.pkg = pkg;
        SystemCache.get().setSaxLoader((Object)null);
}
```

Listing 2.7: Decompiled version

If we look at Listing 2.7, showing the decompiled code for the construct, it is possible to notice that cast to *(Object)* is added to the parameter of the function *setSAXLoader*. This does not allow for the matching of the leaves when the ASTs are compared, leading to assess the construct as vulnerable.

This was, however, only one of the problems leading to a wrong assessment for this bug. Let us analyze the method *parseContentTypesFile(InputStream)*, for which I will only show the modifications happened in the commit fixing the vulnerability, in Listings 2.8 and 2.9.

```
try {
        SAXReader xmlReader = new SAXReader();
        Document xmlContentTypetDoc =
             xmlReader.read(in);
} catch (DocumentException e) {
        throw new InvalidFormatException(
             e.getMessage());
}
```

Listing 2.8: Vulnerable version

```
try {
        Document xmlContentTypetDoc =
             SAXHelper.readSAXDocument(in);
} catch (DocumentException e) {
        throw new InvalidFormatException(
             e.getMessage());
}
```

Listing 2.9: Fixed version

The overall change expected is:

```
Update: Document xmlContentTypetDoc = xmlReader.read(in);
Delete: SAXReader xmlReader = new SAXReader();
```

The actual overall change computed by the algorithm is much longer[7]:

```
...
Insert: i ++;
Delete: Element element = (Element) elementIteratorOverride.next();
Update: Document xmlContentTypetDoc = xmlReader.read(in);
Update: String extension = element.attribute(EXTENSION_ATTRIBUTE_NAME).getValue();
Delete: PackagePartName partName = PackagingURIHelper.createPartName(uri);
...
Delete: SAXReader xmlReader = new SAXReader();
...
```

Indeed the computed overall change contains the changes needed in order to apply the fixes, but it only contains a lot of changes that are not security related at all. Having the overall change not containing the security fixes is a potential source of error, because an intersection between overall change and defective to test could be found even though the security fixes are not contained in the code, but only the other unrelated changes.

CVE-2016-3081

NVD description: *This CVE affects library Apache Struts 2.x before 2.3.20.2, 2.3.24.x before 2.3.24.2, and 2.3.28.x before 2.3.28.1. It allows, when Dynamic Method Invocation is enabled, remote attackers to execute arbitrary code via method: prefix, related to chained expressions.*
Since version Apache Apache Struts 2.3.28.1 is fixed, I included it as dependency in the test application and run version check, expecting to find fixed as output. This was however not the case, and version check was assessing such version as vulnerable. we can have a look at method *setValue()*

[7] I avoided inserting all the changes since they were 38, but I only included some of them in order to give an idea of the problem.

in Listings 2.10 and 2.11.

```
public void setValue(final String name, final
     Map<String, Object> context, final Object
     root, final Object value) throws
     OgnlException {

    compileAndExecute(name, context, new
        OgnlTask<Void>() {
      public Void execute(Object tree) throws
          OgnlException {
        if (isEvalExpression(tree, context))
            {
              throw new OgnlException("Eval␣
                   expression␣cannot␣be␣used␣as␣
                   parameter␣name");
            }
        Ognl.setValue(tree, context, root,
            value);
        return null;
      }
    });

}
```

Listing 2.10: Original source code

```
void setValue(final String name, final
     Map<String, Object> context, final Object
     root, final Object value) throws
     OgnlException {
        this.compileAndExecute(name, context,
             (OgnlUtil.OgnlTask<Object>)new
             OgnlUtil.OgnlUtil$1(this,
             (Map)context, root, value));
    }
```

Listing 2.11: Decompiled source code

For this CVE it is clear how the decompilation process is not always reliable and it can produce output very different from the original one. As we can notice from the previous Listings 2.10 and 2.11 the decompiler in use is not able to deal with anonymous classes.

CVE-2009-2625

NVD description: *This CVE affects library Xerces2 JAVA. The class XMLScanner.java in Apache Xerces2 Java allows remote attackers to cause a denial of service (infinite loop and application hang) via malformed XML input, as demonstrated by the Codenomicon XML fuzzing framework.* Since version Apache Xerces2 2.10.0 is fixed, I included it as dependency in the test application and run version check, expecting to find fixed as output. This was however not the case, and version check was assessing such version as vulnerable. We can have look at the method *scanExternalID(String[],boolean)* in Listings 2.12 and 2.13.

The decompiled code differs from the original in because of:

- Symbols replaced with ASCII code (decimal);
- Variable names get lost;
- Part of if structed changed (else becomes if not);
- Identifiers *final* and *this* are added to some of the variables, whereas they were not in the original source code.

All these differences do not allow to give the correct assessment, marking the construct as vulnerable whereas it is fixed.

CVE-2012-6153

NVD description: *This CVE affects library Apache Commons HttpClient before 4.2.3. The class AbstractVerifier does not properly verify that the server hostname matches a domain name in the*

```
do {
    fStringBuffer.append(ident);
    int c = fEntityScanner.peekChar();
    if (XMLChar.isMarkup(c) || c == ']') {
        fStringBuffer.append(
             (char)fEntityScanner.scanChar());
    }
    else if (XMLChar.isHighSurrogate(c)) {
        scanSurrogates(fStringBuffer);
    }
    else if (isInvalidLiteral(c)) {
        reportFatalError(
             "InvalidCharInSystemID",
               new Object[] {
                    Integer.toHexString(c) });
        fEntityScanner.scanChar();
    }
} while (fEntityScanner.scanLiteral(quote,
     ident) != quote);
```

Listing 2.12: Fixed version

```
do {
    this.fStringBuffer.append((XMLString)o);
    final int peekChar2 =
         this.fEntityScanner.peekChar();
    if (XMLChar.isMarkup(peekChar2) ||
         peekChar2 == 93) {
        this.fStringBuffer.append(
        (char)this.fEntityScanner.scanChar());
    }
    else if
         (XMLChar.isHighSurrogate(peekChar2)) {
        this.scanSurrogates(this.fStringBuffer);
    }
    else {
        if (!this.isInvalidLiteral(peekChar2))
             {
            continue;
        }
        this.reportFatalError(
             "InvalidCharInSystemID", new
             Object[] {
             Integer.toHexString(peekChar2) });
        this.fEntityScanner.scanChar();
    }
} while
     (this.fEntityScanner.scanLiteral(peekChar,
     (XMLString)o) != peekChar);
```

Listing 2.13: Decompiled source code

subject's Common Name (CN) or subjectAltName field of the X.509 certificate, which allows man-in-the-middle attackers to spoof SSL servers via a certificate with a subject that specifies a common name in a field that is not the CN field.
Since version Apache HttpClient 4.2.3 is fixed, I included it as dependency in the test application and run version check, expecting to find fixed as output. This was however not the case, and version check was assessing such version as vulnerable. We can have a look at method *getCNs(X509Certificate)* in Listings 2.14 and 2.15:

```
while(st.hasMoreTokens()) {
  String tok = st.nextToken().trim();
  if (tok.length() > 3) {
    if (tok.substring(0,
         3).equalsIgnoreCase("CN=")) {
      cnList.add(tok.substring(3));
        }
  }
}
```

Listing 2.14: Fixed version

```
while (st.hasMoreTokens()) {
    final String tok = st.nextToken().trim();
    if (tok.length() > 3 && tok.substring(0,
         3).equalsIgnoreCase("CN=")) {
        cnList.add(tok.substring(3));
    }
}
```

Listing 2.15: Decompiled source code

This is another problem related to the decompiler, since the nested ifs are merged in a single 'if'. This difference does not allow to give the correct assessment, marking the construct as vulnerable whereas it is fixed.

2.2.2 Problems overview

The main problems emerged during the analysis are:

1. Decompiler related
   (a) Adding of identifiers 'final' and 'this':
   This problem arises when the decompiler adds the keywords 'final' or 'this' to the code, whereas they were not in the original source code. When it happens, nodes in ASTs don't match anymore because in ChangeDistiller one of the criteria for matching is the string similarity of their values.

(b) Cast to 'Object': This problem arises when the decompiler adds cast to Object to the code, whereas it was not in the original source code. When it happens, nodes in ASTs don't match anymore because in ChangeDistiller one of the criteria for matching is the string similarity of their values.

(c) Constants:
During the compilation/decompilation process, values of the constants in the code are replaced with their runtime value. When analyzing the source code of the fix, the same constant can have a different value between decompiled and original source code, thus making impossible, for ChangeDistiller, the identification of the node representing it.

(d) 'If' nesting and changes in 'if' conditions:
Because of the process of optimization during both compilation and decompilation phase, the value of 'if' conditions are changed (preserving the logical flow of the program). This does not longer allow for matching the nodes of the ASTs representing the code.

(e) 'For' using an iterator turned into 'while':
Because of the process of optimization during both compilation and decompilation phase, a 'for' that it's using an iterator is changed into its equivalent 'while'. This does not longer allow for matching the nodes of the ASTs representing the code.

(f) Nested and anonymous classes.
The decompiler in use does not correctly handle nested and anonymous classes, leading to big inconsistencies with the original code. When this happens, ASTs to be compared are really different from each other, making it impossible the comparison.

2. Edit scripts related

(a) Mismatched leaves by Change Distiller:
When computing the overall change for the patch, in order to see which are the changes that fix the vulnerability, the edit script between the versions before and after the fixing commit are compared. In order to extract the changes, the algorithm finds the matching between the nodes in the two trees representing the two versions and then, given the computed matching, finds the minimum edit script that transforms the first tree in the second one. In the above mentioned process, a very important step is the matching of the leaves: a wrong matching for two leaves would be propagated up to their parents, bringing in very big differences between the real edit script and the wrongly computed one. In ChangeDistiller two leaves are matched if they have the same label and if their similarity, computed using Levenshtein distance, is above a certain threshold. However the parent is not taken into account when computing the similarity of the leaves, which leads leaves with same label and same value to be matched to the wrong father.

(b) Construct modified on multiple commits, some of them not security related:
The overall change is computed between the version before the first commit and the version after the last commit fixing the vulnerability (for a single construct). If the same construct is modified in more then one commit, and some of these commits are not security related, the overall change would contain all of these changes, whereas in the change list for the construct only security related changes are contained.

2.3 Solutions overview and implementation

For the problems emerged during the analysis and listed in 2.2.2, I will describe my proposal for possible solutions and implementation.

Adding of identifiers 'final' and 'this' by the decompiler [1a]

An easy solution to problem 1a would consist in removing all 'final' and 'this' identifiers, but this might might not be good if such identifiers are part of the security fix. A better solution is to check

whether the original source code contains such identifiers and, if not, when comparing the nodes of ASTs for equality check to remove these identifiers from the AST of decompiled code. This is the solution that I actually implemented, and which helped in correctly assessing CVE-2011-1498.

Cast to 'Object' [1b]

An easy solution to problem 1b would consist in removing all the cast to 'Object', but this might might not be good if such identifiers are part of the security fix (it is highly unlikely, but better be conservative). A better solution is to check whether the original source code contains such cast and, if not, when comparing the nodes of ASTs for equality check to remove the cast from the AST of decompiled code. This is the solution that I actually implemented, and which helped in correctly assessing CVE-2014-3529.

Values of constants changed by the decompiler [1c]

Since the decompiler in use changes the literal value of the constants with their runtime value, it is necessary to look for the class where the constant is initially defined and use the same value when matching nodes in ASTs. After having it implemented, it helped in correctly assessing CVE-2011-1498 is obtained.

Mismatched leaves by ChangeDistiller [2a]

In order to extract the changes, version check finds the matching between the nodes in the two trees representing the two versions and then, given the computed matching, finds the minimum edit script that transforms the first tree in the second one.

In the above mentioned process, a very important step is the matching of the leaves: a wrong matching for two leaves would be propagated up to their parents, bringing in very big differences between the real edit script and the wrongly computed one. In ChangDistiller two leaves are matched if they have the same label and the similarity for their values, computed using Levenshtein distance [15], is above a certain threshold. What if two leaves have the same value and the same label type? In the example shown in Figure 2.5 below, leaves have the same label type (return statement) and the same value (""), so Change Distiller will simply match the first one it finds.

The original algorithm does not take into account parents of the leaves in order to compute

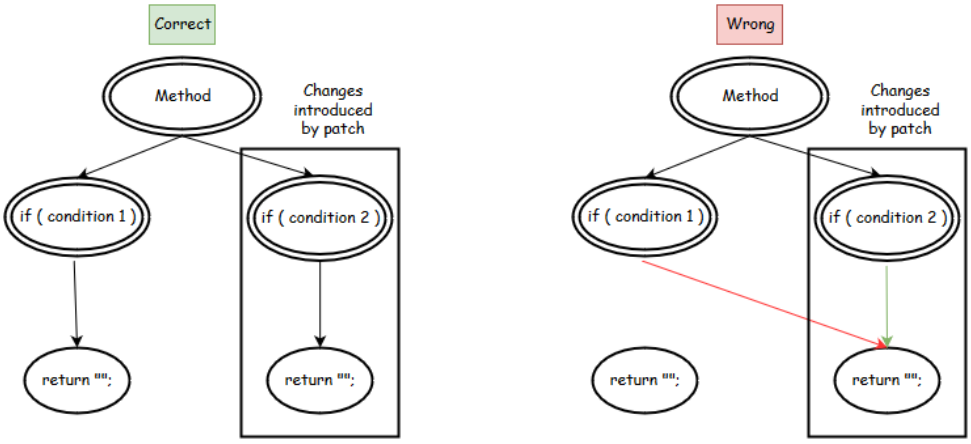


Figure 2.5: Correct and wrong matching of leaves having the same value.

the similarity between their values. In fact the original algorithm does not care about this case since, in the end, the second tree would be obtained anyway, even switching the leaves between their parents. For the purpose of the security fixes check, the final AST is not the interesting part, instead the changes applied in order to get to it. When leaves have not the same label and

their similarity is different from 1 (1 indicates equality), it is not necessary to interfere with the algorithm since there is not risk for them to be matched wrongly. Instead, when their similarity value is equal to 1, we might want to consider also the similarity of their parents in order to check if they are exactly the same leaf. In case two leaves have similarity equal to 1, a possible approach is to penalize their similarity by subtracting their parent similarity. By applying this approach similarity between two leaves sharing the same parent will not be decreased, yes otherwise.

Pseudocode

```
if ( similarity == 1 ) {
   // try to penalize leaves with no common parent, but having same value
   double parentSimilarity = fLeafGenericStringSimilarityCalculator.calculateSimilarity(
          x.getParent().toString(),
          y.getParent().toString());
   similarity = similarity - ( 1 - parentSimilarity );
}
```

By adding this fix, the correct overall change for CVE-2011-1498 is obtained.

Construct modified on multiple commits, some of them not security related [2b]

The overall change normalization is necessary when more than one commit is made in order to fix a vulnerability, and in the meantime other commits that are not security related are made on the same method. Let us consider the example in Figure 2.6:
Commit 1 is the first fix for the vulnerability and commit 4 is the last fix, while commits 2 and 3

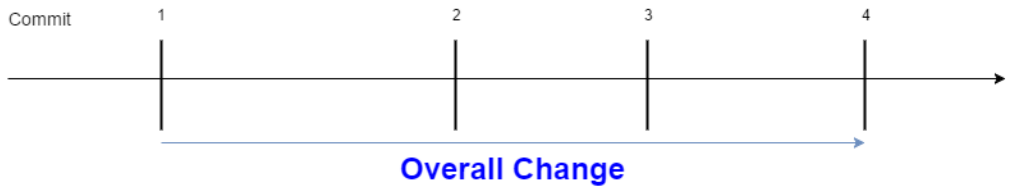


Figure 2.6: Timeline of commits performed on a single construct.

are not related to the fix. Commits that are not fixing the vulnerability are not saved in the backend database when performing the patch analysis for the vulnerability since they are not interesting for our purpose of finding the security fixes in the analyzed archive. When, later in time, version check is run on an archive potentially affected by the vulnerability, it computes the overall change between the first and last (fixing) commit available. Although this is correct from the edit-script point of view, it can potentially introduce source of error: in the extreme case we might have an archive for which edit scripts intersections contain all the not security related fixes, but not the security related ones. In such a case we would have the archive assessed as fixed even though it is vulnerable. The wrong assessment can also happen in the opposite direction: an archive might be assessed as vulnerable whereas it is fixed. In fact, if the not security related changes are removed from the archive, think of refactoring of the code, there will be an intersection between the overall change and the edit script test to fixed. In order to fix this, I made this assumption:

Assumption 2. *In order to assess the vulnerability state of a construct, it is not interesting to find all the changes happened to the construct, but only the ones that are relevant for fixing the vulnerability.*

Starting with this assumption, we have to :

1. Consider all the commits available for that vulnerability;
2. Find out if, between any couple of them, there are commits that are non relevant for the fix;
3. If any, remove the changes introduced by these commits from the overall change, obtaining the correct overall change.

Pseudocode

```
commitsSet = {ordered set of all commits fixing the vulnerability}

removeIntermediateCommits(commitsSet)
{
   // get the overall change between first an last commit
   overallChange = editScript(first(commitsSet),last(commitsSet))
   // iterate all over the commits
   for i in [0:commitsSet.length] :
      // if the sets of changes necessary to go from commit i to commit i+1 is not empty
      // there is at least one not-fixing commit between them
      if (commitsSet[i].after != commitsSet[i+1].before):
         // compute the delta not-fix, that is the edit script necessary to go from
         // commit i to commit i+1
         deltaNotFix = getChanges(commitsSet[i].after, commitsSet[i+1].before)
         // iterate all over the changes in overallChange and deltaNotFix and remove the ones in
              common
         for ( change in overallChange ):
            for ( changeNotFix in deltaNotFix):
               if ( change == changeNotFix ):
                  overallChange.remove(change)
   return overallChange
}
```

After applying the new algorithm, correct overall change for CVE-2014-3529 is computed, and bug is assessed correctly.

Unsolved problems

Though I identified and implemented solutions for several problems, still some remains unsolved, namely:

- If nesting and changes in if conditions [1d]
- 'For' using an iterator turned into 'while' [1e]
- Nested and anonymous classes [1f]

One direction to investigate in order to solve the problem above starts by changing the decompiler in use. However, as the solution described above, they would be custom solution for the identified issues that cannot provide any assurance for the general problem.

2.4 Evaluation

In this section I will evaluate the results obtained with version check before and after having applied the improvements. My general approach for evaluation has been:

1. Choose a set of bugs for widely used Apache libraries;
2. For each bug, manually verify (through NVD) which library is affected and which versions for the library are the last vulnerable and the first fixed;
3. Run version check for each bug and both libraries and verify the assessment;
4. Run version check again for each bug, this time using the latest available release for the affected library.

By analyzing the results obtained in Table 2.1, we can notice that before applying the improvements the number of wrong assessments is 18 over a total of 39 analyzed archives (46%).

Bug	Assessment result Vulnerable version	Assessment result Fixed version	Assessment result Latest version
CVE-2009-2625		X	X
CVE-2011-1498		X	X
CVE-2012-0838		X	X
CVE-2012-2098			
CVE-2012-6153		X	X
CVE-2013-2186			
CVE-2014-0050			
CVE-2014-3529		X	X
CVE-2014-3574		X	X
CVE-2014-3577			X
CVE-2016-2162			X
CVE-2016-3081		X	X
CVE-2016-3082		X	X

Table 2.1: Correctness of results obtained before applying improvements. (X indicates wrong assessment)

Bug	Assessment result Vulnerable version	Assessment result Fixed version	Assessment result Latest version
CVE-2009-2625		X	X
CVE-2011-1498			
CVE-2012-0838		X	X
CVE-2012-2098			
CVE-2012-6153		X	X
CVE-2013-2186			
CVE-2014-0050			
CVE-2014-3529			X
CVE-2014-3574			X
CVE-2014-3577			X
CVE-2016-2162			X
CVE-2016-3081		X	X
CVE-2016-3082			

Table 2.2: Correctness of results obtained after applying improvements. (X indicates wrong assessment)

After applying the improvements, as shown in Table 2.2, the number of wrong assessments decreases down to 12 over 39 (31%).

In order to evaluate the results, I had to manually analyze the output of version check for each analyzed archive and at the same time check on NVD's website the expected assessment for that archive. This is made necessary since some problems with the decompiler could not be solved, thus not making reliable the assessment given by version check. Having this unreliability breaks the very foundation of the approach, that is to give automatic assessments with no need for manual review.

From the results of the assessment when using the latest available release it is interesting to notice how, despite the improvements, the assessment keeps being wrong. This can be due to the fact that, since the approach is based on the comparison of the source code, when a library is updated in releases after the one where the security fix is introduced, the new changes might interfere with the AST representation, no longer allowing to compare the ASTs and find the changes introduced by the fix. It is important to notice how the quality and not the quantity of changes matters, in fact a single line of code might completely change the way edit script is computed between two versions of the source code. Let us think of a simple example, with a single if condition added and which body wraps all the changes introduced by the security fix, that is *sanitize(y)*, in Listings 2.16

and 2.17:

```
construct sampleConstruct(int y){
       int x;
       sanitize(y);
       x = y+1;
       return x;
}
```

Listing 2.16: Fixed version

```
construct sampleConstruct(int y){
       int x;
       if ( configuration() ){
              sanitize(y);
              x = y+1;
       }
       return x;
}
```

Listing 2.17: New version

When edit scripts between the ASTs of the two versions are computed, in the fixed version *sanitize(y)* will have the root method as father, whereas in the new version will have the if statement as father. Since the fix consists of inserting *sanitize(y)* with root method as father, the insert of *sanitize(y)* with if statement as father is not seen as fixing.

The previous example makes clear how even a single line of code can make a big difference in assessing the vulnerability state of the archive. Then how is it possible to trust the results given by version check? In order to answer this question, it is necessary to classify the results given by version check as reliable or unreliable. The idea is to find a set of metrics that allows to compute the distance of the archive under analysis respect to the one where the fix for the bug was first introduced, and verify that there is correlation between the distance and the reliability of the assessment.

2.5 Metrics

Open source libraries are subject to changes during their lifetime, as it is in general normal for code. A first version for a library can look very different from the latest available release, since new software is added, deleted or modified in order to improve quality, provide new features and fix bugs. Security bugs, similarly for other kind of bugs, typically affect only a small portion of the code of the library and only require such portion of code to be modified in order to be fixed[8]. Another aspect to be taken into account is that bug fixes may change over time as the code including the fix may be subject to refactoring.

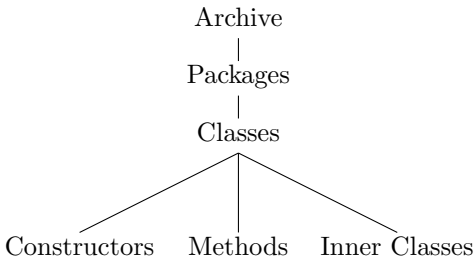
The idea at the core of version check is to look for the containment of the fix in its original form, i.e., as it was applied when the bug was patched for the first time. As a result, if the code is not found anymore because of a refactoring, version check wrongly concludes that the archive under analysis is vulnerable, whereas it is fixed. As version check cannot be always applied, we aim at determining a set of metrics able to establish the closeness of archives. Then check version can only be applied in case archives are close, assuming that this is an indication that the archive did not undergo a refactoring.

With the term metrics I intend a set of numbers that are representative for an archive and should, ideally, allow for the identification of the difference(s) between two different versions of the same archive.

2.5.1 Simple Metrics

Any Java archive is usually structured in this hierarchical way:

[8]This has emerged during the analysis of the various CVEs, where usually only a few classes were interested by the fix.



A simple version for the metrics can be computed according to the hierarchical representation of the archive starting from the similarity between the classes and going up in the hierarchy up to archive level. Following the above mentioned criteria, a simple algorithm can be:

- Within the same class, count the number of methods and constructors that are common to both the archives. I only considered methods and constructors since they are the elements of the change list of a bug that are analyzed by version check;
- At a package level, use the similarity computed for the classes in order to compute the similarity between the packages;
- At archive level, use the similarity computed for the packages in order to compute the similarity between the archives.

Having this simple version implemented, I ran it on several sets of archives in order to study the variation of the similarities at different levels, trying to find some interesting correlations.
Full tables representing the results are available in Appendix B.2.

Pseudocode

```
firstPackages = packages comprised in first archive;
secondPackages = packages comprised in second archive;

archiveSimilarity=0;
for package p in firstPackages:
       packageSimilarity=0;
       if p in secondPackages:
              // retrieves all the classes comprised in package p for 1st and 2nd archive
              classesFirst = p.getClasses()
              classesSecond = secondPackages.get(p).getClasses()
       for class c in classesFirst:
              if c in classesSecond:
              classSimilarity = computeClassSimilarity(c, classesSecond.get(c));
              packageSimilarity += classSimilarity;
        totalClasses = classesFirst.size() + classesSecond.size();
   packageSimilarity = (packageSimilarity*2)/totalClasses;
   archiveSimilarity += packageSimilarity;
totalPackages = firstPackages.size() + secondPackages.size()
archiveSimilarity = (archiveSimilarity*2)/totalPackages;

function computeClassSimilarity(class from, class to):
       firstMethods = from.getAllMethods();
       secondMethods = to.getAllMethods();
   commonMethods=0;
   for method m in firstMethods:
       if m in secondMethods:
              commonMethods++;
   totalMethods = firstMethods.size()+secondMethods.size();
   return (commonMethods*2)/totalMethods;
```

2.5.2 Dependency Finder

Dependency Finder is a suite of tools for analyzing compiled Java code. At the core is a powerful dependency analysis application that extracts dependency graphs and mines them for useful information.[9]
Although the purpose of the tool itself it's not meant to be used as comparison between different versions of the same archive, information it produces might be interesting for analyzing such differences. In fact, among the many features of this open source library, we can also find one called *OO Metrics*, standing for object oriented software metrics. These metrics are meant to evaluate the quality of the software, and the most interesting ones (for my research) are:[10]

- Method level:
  - Local Variables (LVAR) : Number of local variables used by the method;
  - Outbound Intra-Class Feature Dependencies (OICF) : Methods and fields within the same class that this method depends on;
  - Outbound Intra-Package Feature Dependencies (OIPF) : Methods and fields within the classes of the same package that this method depends on;
  - Outbound Intra-Package Class Dependencies (OIPC) : Classes within the same package that this method depends on;
  - Outbound Extra-Package Feature Dependencies(OEPF) : Methods and fields in other packages that this method depends on;
  - Outbound Extra-Package Class Dependencies (OEPC) : Classes in other packages that this method depends on.
- Class level:
  - Methods (M) : Number of methods in this class;
  - Attributes (A) : Number of attributes in this class;
  - Inner Classes (IC) : Number of inner classes in this class;
  - Subclasses (SUB) : Number of direct subclasses of this class;
  - Depth of Inheritance (DOI) : How many hops there are between the top of the inheritance hierarchy and this class;
  - Outbound Intra-Package Dependencies (OIP) : Classes of the same package that this class depends on;
  - Outbound Extra-Package Dependencies (OEP) : Classes of other packages that this class depends on.
- Package level:
  - Classes (C) Number of classes in this package;
  - Methods (M) : Number of methods in this package;
  - Attributes (A) : Number of attributes in this package;
  - Subclasses (SUB) Number of direct subclasses of this package;
  - Depth of Inheritance (DOI) : How many hops there are between the top of the inheritance hierarchy and this package.

I ran the tool on several sets of archives in order to study the variation of the similarities at different levels, trying to find some interesting correlations.
Full tables representing the results are available in Appendix B.1.

[9] From official website http://depfind.sourceforge.net/

[10] Re-adapted from the official wiki http://depfind.sourceforge.net/Manual.html

2.5.3 Results evaluation through a case study

In this section I will explain the metrics computed using both the simple metrics and dependency finder for the archive *httpcomponents-httpclient* at class, package and method level relative to the method *org.apache.http.conn.ssl.AbstractVerifier.getCNs(X509Certificate)*. I chose these data because version check fails in assessing the archive *org.apache.httpcomponents-httpclient-4.5.2* for bug CVE-2014-3577. This archive should be fixed, and its first fixed version is *4.3.5* that is correctly assessed by version check.

We can have a look at fixed and tested version for this method in Figure 2.7 and 2.8.

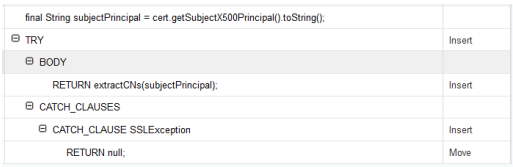


Figure 2.7: Fixed body, version 4.3.5, including on the right column the type of change required to apply the fix.

Figure 2.8: Tested body, version 4.5.2.

Because of the logic of version check, the tested body of Figure 2.8 is assessed as vulnerable. In fact:

1. One of the changes that fixes the vulnerable is the inserting of *RETURN extractCNs(subjectPrincipal)*;
2. The tested body does not contain *RETURN extractCNs(subjectPrincipal)*, therefore the edit script tested to fix will not be empty;
3. When intersecting the edit scripts overall change and from tested to fix, the intersection will not be empty since they both contain *RETURN extractCNs(subjectPrincipal)*;
4. Since previous intersection is not empty, the version under test is assessed as vulnerable.

This is one example where check version cannot work. In the following we will apply the metrics to this example to check whether they are suited for establishing the closeness of the archive. Let us analyze the results of the metrics for this archive, I will use the following three versions:

- Version 4.3.5, which is fixed;
- Version 4.5.2, which we know is fixed but for which, as shown above, check version cannot work;
- Version 4.2.5, which is vulnerable, and for which check version works.

In the following we will refer to the versions above as 'fixed','test', and 'working', respectively.

Let us start with the simple metrics[11]. The results at archive level are shown in Table 2.3.

Version 'test' seems to be more similar to version 'fixed' than version 'working' is. In fact, as shown in Table 2.3, similarity between version 'fixed' and 'working' is 81%, whereas similarity between 'fixed' and 'test' is 92%. Since the archive level might be too coarse-grained, we computed the metrics at package level (see Table 2.4).

As for the archive level, the result is not the one we would expect, since test version is still closer to fixed version than working version is. In fact, as shown in Table 2.4, similarity between

	working	test
fixed	81%	92%

Table 2.3: Similarity at archive level between version 'fixed' and, respectively, 'working' and 'test'.

	working	test
fixed	63%	85%

Table 2.4: Similarity at package level between fixed, working and test version.

version 'fixed' and 'working' is 63%, whereas similarity between 'fixed' and 'test' is 85%. The similarity at class level is shown in Table 2.5.

The results at class level reflect the expectations, however the values are still close. In fact, as shown in Table 2.5, similarity between version 'fixed' and 'working' is 88%, whereas similarity between 'fixed' and 'test' is 72%. Other questions arise: is it 72% low enough to draw a conclusion? Is it the class level a good level of granularity?

Let us consider a simple (mock) example:

- As part of a security fix, method with name M is deleted from class C in the fixed version. Deletion of method M is the only change performed in class C;
- Later on, in the same class C, a new method with the same name M is added again but having a completely different body. No other methods or constructors have been added in the meantime;
- No matter how many methods/constructors are in class C, the similarity between these three versions of class C will always be the same;
- Version check wrongly assesses the latest released version as the method M (that was deleted in the fix) exists.

Even though the example is very specific, it clearly shows that if an archives undergoes small changes over different version, then the simple metrics cannot be applied for drawing reliable conclusions on the similarity. If we think of methods that are modified as part of security fix, and then modified again because of refactoring, still we would get very high similarity values, but version check will not be able to perform a correct assessment.

In order to consider the changes for method bodies, I used the metrics computed by dependency finder. I will keep on considering versions 4.2.5, 4.3.5 and 4.5.2 for the analysis of the results[12]. As an example, I will consider[13]:

- At package level, metric 'A', which values are shown in Table 2.6.
- At class level, metrics 'M' and 'A', which values are shown in Table 2.7.
- At method level, metrics 'LVAR' and 'OEPF', which values are shown in Table 2.8.

For all the three levels of analysis, version 'test' seems to be closer to version 'fixed' than version 'working' is. No correlation that is either good or obvious can be drawn by looking at this set of metrics.

[11] Full tables with results of the simple metrics can be found in Appendix B.2

[12] Full tables with results of Dependency Finder can be found in Appendix B.1

[13] See Section 2.5.2 for the meaning of the metrics' names.

	working	test
fixed	88%	72%

Table 2.5: Similarity at class level between fixed, working and test version.

	fixed	working	test
Metric A	2.25	1.3	2.55

Table 2.6: Value of metric 'A' at Package level between fixed, working and test version.

	fixed	working	test
Metric M	16.0	13.0	13.0
Metric A	2.0	2.0	2.0

Table 2.7: Value of metrics 'M' and 'A' at class level between fixed, working and test version.

	fixed	working	test
Metric LVAR	6.0	3.0	4.0
Metric OEPF	2.0	15.0	2.0

Table 2.8: Value of metrics 'LVAR' and 'OEPF' at method level between fixed, working and test version.

Even though this is only one example, similar results were obtained for the other archives considered. Due to the fact that different versions undergo a limited number of modifications, quantitative metrics are not sufficient for establishing the similarity as different changes compensate each other.

Though the metrics cannot be used to establish whether version check can be applied, they clearly showed that the similarity between the same class over different release is relatively high and stable. The same behavior can be noticed if looking at results obtained using dependency finder at method level.

The conclusion is that classes/methods are quite stable over time and they are changed as few as possible. This especially happens when a security fix needs to be introduced. This can be very useful and it opens to new possibilities: if methods/classes are so stable over time, is it possible to look for actual equality between the methods under test and the vulnerable/fixed ones?

Chapter 3

Patch evaluator

Nowadays released versions of OSS libraries available through software repositories (e.g., Maven central [16]) do not provide an explicit reference to the revision of the VCS repository based on which they were build. Still, dealing with open source, it is often the case that the source code is available for download. As an example, for each artifact available for download in Maven central, both the archive of binaries and source code may be available. Since source code is available both from VCS and software repositories, it might be possible to assess whether libraries are vulnerable by simply comparing source code. Since both source code for commits and released library is available, might it be possible to rediscover such missing link and automatically assess a large number of libraries by directly comparing their source code?

In this chapter I will explore the viability of this idea and describe the approach I developed. I will show four incremental approaches, starting from the basic idea behind them and then showing how they can be combined in order to get the highest possible number of assessments and the highest confidence in the results as well.

3.1 Preliminary analysis

At the basis of this approach is the availability of source code. Though for OSS libraries this is usually available, this is not always the case. As an example, when the library development switches to a new VCS repository, the older revisions of the source code may not be available any longer. An example of this is shown in Figure 3.1.

GroupId	ArtifactId	Version	Updated	Download
org.apache.tomcat	catalina	6.0.18	05-Aug-2008	pom jar

GroupId	ArtifactId	Version	Updated	Download
org.apache.tomcat	tomcat-catalina	8.5.9	05-Dec-2016	pom jar sources.jar

Figure 3.1: Example of two different libraries available on Maven central, respectively with and without sources available.

As a preliminary analysis, in order to verify in how many cases it is possible to retrieve the source code, I wrote a simple algorithm performing these steps:

1. Get the list of all the bugs from vulas backend;
2. For each of the bugs, get the list of the libraries containing the vulnerable code;
3. For each of the libraries containing the vulnerable code, download the sources(if any);
4. Compute the percentage of distinct libraries whose sources are available over the total.

The conclusion is that over a total of 981 libraries[1], 835 comes with sources. This means that the sources are available for 85% of the library. As a result an approach based on source code analysis can be applied on a relevant share of libraries.

3.2 Assessment by equality

When a bug is fixed, a subset of the constructs belonging to the vulnerable library are modified in order to solve the vulnerability, this subset is the so-called change list[2]. Having the change list allows, on one side, to understand the changes that are necessary to fix the vulnerability and, on the other side, to identify the last version of the vulnerable construct and the first version of the fixed one. If the very same source code can be found in the source code attached to a released library, then it is possible to conclude that the vulnerable code is contained in its vulnerable (fixed, resp.) version.

To fix a bug may require more than one commit in a VCS repository, where each commit may involve modification to several constructs. Thus the comparison described above has to be done for each modified construct. Moreover, if the same construct has been touched in several commits, it is necessary to find the latest vulnerable code (corresponding to the code before the last commit) and the latest fixed code (corresponding to the code introduced by the last commit). It is therefore possible to analyze the code of the archive under analysis for each of its constructs and obtain:

- If the code under test is exactly equal to the vulnerable code, it means that the version of the construct under test is vulnerable;
- If the code under test is exactly equal to the fixed code, it means that the version of the construct under test is fixed;
- If the code under test is neither exactly equal to the vulnerable nor to the fixed version, it is not possible to automatically assess whether the construct under test is vulnerable or fixed.

If for each of the constructs it is possible to find the equality, then it is possible to assess the state of the archive. Of course this cannot always happen, since the constructs from the change list represent a snapshot of the construct body at the time the fix was first applied and thus they may differ from previous and later releases. Moreover, in the very same release there might be equality only for a subset of constructs. Let us look at a simple example:

- Construct c1 is modified as part of the fix for a certain bug in commit 1;
- Construct c2 is modified as part of the fix for the bug in commit 2, but also construct c1 is modified again in commit 2, this time not related to the fix;
- The change list will contain only the version for construct c1 in commit 1 and construct c2 in commit 2 because these are part of the fix;
- The released archive will include, for both the constructs c1 and c2, the version released with commit 2;
- When attempting to match source code found in the released archive, there will be exact match only for construct c2.

This example is shown in Figure 3.2.

By analyzing all the constructs from the change list found in the archive, there are six possible outcomes:

[1] A distinct library is defined by the triplet Group,Artifact,Version according to Maven representation.

[2] More details about the change list can be found in Section 2.1.1

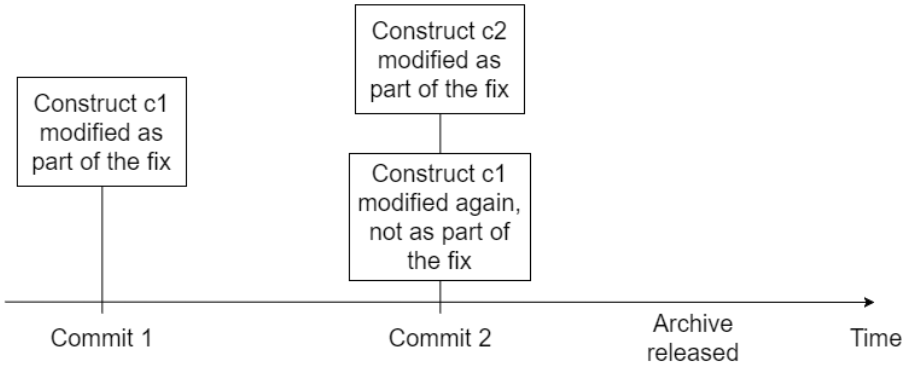


Figure 3.2: Example of constructs modified in 2 subsequent commits.

1. All the constructs from the change list found in the archive are exactly equal to their respective vulnerable version;

2. All the constructs from the change list found in the archive are exactly equal to their respective fixed version;

3. Some of the constructs from the change list found in the archive are exactly equal to their respective vulnerable version, no decisions can be taken on the remaining ones;

4. Some of the constructs from the change list found in the archive are exactly equal to their respective fixed version, no decisions can be taken on the remaining ones;

5. Some constructs from the change list are found equal to their fixed version, while others are found equal to their vulnerable version (and possibly others are not found equal to any of them). This case should ideally never happen, since it would mean that only partially fixed archives are released, but it needs to be taken in account.

6. For none of the constructs from the change list found in the archive exact equality, neither for respective fixed nor vulnerable version has been found.

In order to cope with all six possible situations, I made the following assumption:

Assumption 3. *If at least one among the constructs from the change list found in the archive is exactly equal to its vulnerable (fixed, resp.) version, and no decision can be taken on the remaining constructs, the archive is vulnerable (fixed, resp.). If both vulnerable and fixed constructs are found, the archive is vulnerable.*

By using Assumption 3, is it possible to say that the assessment is reliable? If at least one construct is vulnerable, it is reasonable to conclude that the archive is vulnerable. However, when not all constructs found are fixed, it is not obvious whether the entire archive is fixed. Assumption 3 is justified since the fixed version of the code for the construct under analysis might not be found exactly equal because of, i.e., further commits (not security related) that modify the construct or removal of logs from the packaged archive. Assumption 3 will be validated by the results obtained in Chapter 4.

Since it takes into account only one archive at a time, the assessment by equality has an important shortcoming:

- No decision can be taken if there is not equality for any of the constructs from the change list.

3.3 Improvement

The goal of the patch evaluator approach is to assess all version of the archive[3] under analysis. Is it possible to combine the results coming from the assessments for all the version of the archive under analysis[4] in order to get a better, and more reliable, result?

If the assessment by equality approach is applied to all the versions of the archive under analysis, then it is possible to obtain, for each construct, equality to both vulnerable and fixed version of the construct for different versions of the library. Doing so, for each construct, it is possible to identify the latest in time released version that is equal to its vulnerable version, and the first version which is equal to its fixed version.[5] The change list is usually containing more constructs, then it is possible to aggregate the information coming from all the constructs in order to get a more accurate idea about which one is the last vulnerable version and which the first fixed version for the archive respect to the bug under analysis. In order to validate this approach, a constraint is added: last vulnerable must always be smaller than first fixed. This validation is necessary since, by combining the information coming from all the constructs, in some of them exact equality might not be found for either vulnerable or fixed version (or both). A possible outcome of the search for last vulnerable and first fixed by combining results coming from different constructs is shown in Figure 3.3.

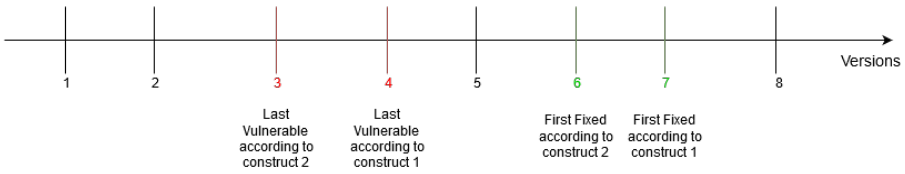


Figure 3.3: By combining the information coming from all the constructs, last vulnerable and first fixed can be identified.

Having last vulnerable and first fixed versions available could allow to assess all the versions released before the last vulnerable as vulnerable, and all the ones released after the first fixed as fixed. There might still be an uncertainty window between the last vulnerable and first fixed version: for the archives released within this time frame no assessment can be done, as shown in Figure 3.4.

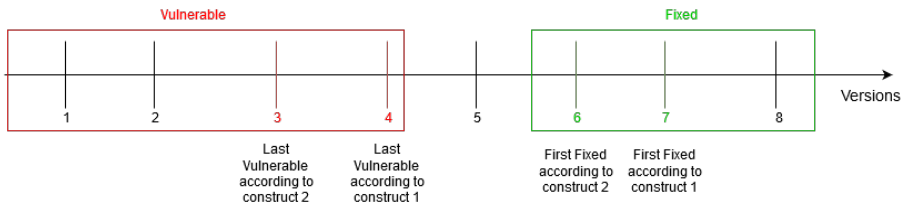


Figure 3.4: By combining the information coming from all the constructs, last vulnerable and first fixed can be identified.

This approach has a very big shortcoming: versions not affected at all by the bug might be wrongly assessed as vulnerable. In order to prove it, let us consider a real world example. NVD description for CVE-2014-0095 states that: *java/org/apache/coyote/ajp/AbstractAjpProcessor.java in Apache Tomcat 8.x before 8.0.4 allows remote attackers to cause a denial of service (thread*

[3]In the following, the term archive means a certain version of the library under analysis.

[4]All the versions of the archive under analysis share the same group and artifact identifier.

[5]Latest in time released version that is equal to its vulnerable version will be shortened with last vulnerable. First version with equality to fix will be shortened with first fixed.

consumption) by using a "Content-Length: 0" AJP request to trigger a hang in request processing.[17] In this case, the vulnerability only affects Apache Tomcat versions from 8.x to 8.0.4, thus all versions belonging to major releases 6, 7 and 9 are not affected. Should the approach consider all the versions below 8.0.4 as vulnerable, a wrong assessment would be done for all the versions belonging to major release 6, 7 and 9. It is then necessary to take into account the major releases archives belong to.

3.3.1 Assessment by major releases

The life cycle of any library usually involves the release of a major version, then the development of a new major release starts while minor versions of the previous (and maintained) major release are released in order to fix the contained bugs.

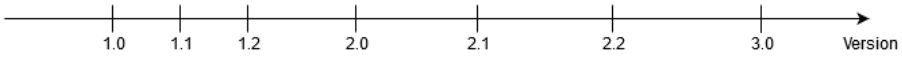


Figure 3.5: Example of versioning for a certain group,artifact. In this case major releases are 1,2 and 3.

This fact can be exploited in order to perform the assessment of versions belonging to the same major release:

- If a version associated with a certain major release is vulnerable, it is possible to assess all the versions belonging to the same major release and released before than the one known to be vulnerable as vulnerable.
- If a version associated with a certain major release is fixed, it is possible to assess all the versions belonging to the same major release and released after than the one known to be fixed as fixed.

These new assessments are reliable since, if a bug is fixed for a certain version in a certain major release, it will not be included again in later versions. Should it happen, the bug will be given a new identifier, so be treated differently (as a new bug).

By comparing versions according to their major release, new uncertainty windows can appear: in fact there might be groups of versions associated to major releases for which there is no information about last vulnerable and/or first fixed. This situation is shown in Figure 3.6.

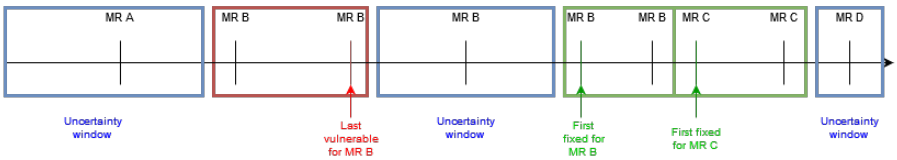


Figure 3.6: Assessment using information about last vulnerable, first fixed, and major release (shortened with MR).

Even though the division by major releases allows to give more reliable conclusions, this approach still does not allow to give an assessment when no version with equality to fixed or vulnerable is found and also to find out when the fix has been introduced.

3.4 Assessment by distance

Although it is not always possible to find equality with vulnerable and/or fixed version, it is still possible to compute the changes that are necessary to transform the version under test in the vulnerable and fixed version. Bigger the number of changes, bigger the distance between the two

versions starting from which changes are computed. For all the constructs in the change list, it is possible to exploit this information in order to check whether the construct in the version under analysis is more distant from its vulnerable or fixed version.

Ideally, the distance to the fixed version is always greater or equal than the distance to the vulnerable version before the fix for the bug is applied, moment after when the situation is reversed, with distance to vulnerable greater than distance to fixed. This situation is shown in Figure 3.7.

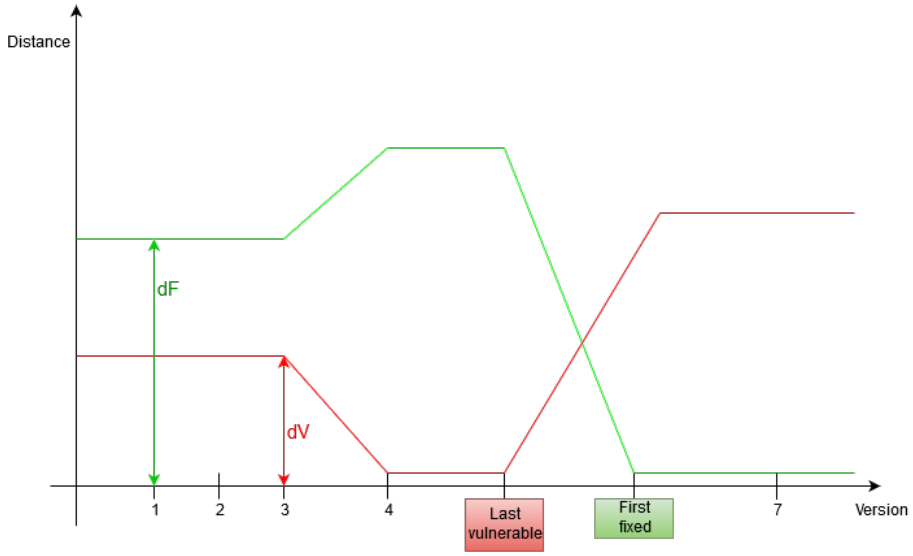


Figure 3.7: Example of representation of distances between version under test and its vulnerable (dV) and fixed (dF) versions.

If distances are represented on a Cartesian plane as lines having the distance as y coordinate and the ordered release for the archive as x coordinate, in the ideal situation there would always be only one intersection between the two lines representing the distances to vulnerable and fixed version, this intersection happening when the fix is introduced. An example of fix is shown in Figure 3.8.

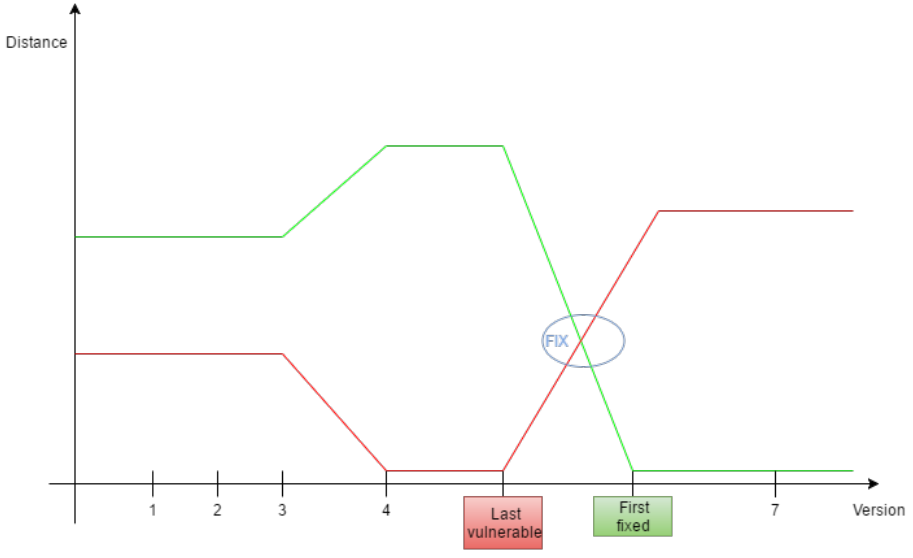


Figure 3.8: Intersection between two lines identifies the moment when the fix is introduced.

Being sure that only one intersection is found, it would be possible to assess the versions released before the intersections as vulnerable, and the one later as fixed. However, the same reasoning valid

for major releases still holds: in order to get more accurate and reliable results, an intersection for every group of versions belonging to the same major release should be found. The information coming from all the constructs of the change list can be exploited with the following approach:

1. Divide the versions by major releases;
2. For all the archives belonging to the same major release compute, for all the constructs, the distances to their vulnerable and fixed versions;
3. Get the intersections for each construct.

In a non ideal situation, the distance from fixed might be lower than distance to vulnerable before the vulnerability is inserted in the code for the construct, and later behave as the ideal case. In such a case, as it is shown in Figure 3.9, a first intersection might be found before the one identifying the fix.

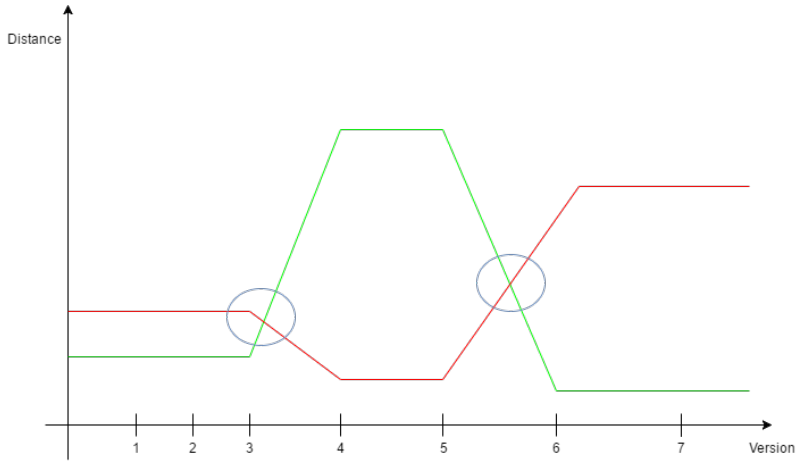


Figure 3.9: Two intersections, first one is not the one identifying the fix.

Cases such this one show the need for a validation of the results found with this approach.

3.4.1 Intersections validation and assessment

In order to get reliable results when using intersections for assessing archives, some rules must be identified in order to distinguish between good and wrong intersections.[6]

An intersection is good if, before it happens, distance to the fixed is bigger than distance to the vulnerable. An example of discrimination between good and wrong intersection is shown in Figure 3.10.

What if more than one good intersection is found within all the constructs? In order to cope with this scenario, it is necessary to think in term of major release division. If plotting the distances on the same chart, a situation like the one shown in Figure 3.11 might happen since different versions for he archive, belonging to different major releases, can be fixed in different moments.

This situation can be avoided by computing intersections separately for each major release. If still more than one intersection is found, how is it possible to establish which one is the good one? From the previous approach, the information coming from the construct with exact equality can be used in order to validate the found intersections. In fact, if information about last vulnerable and

[6]With the term good, I intend intersections that might represent the moment when the fix has been released. On the other hand wrong indicates intersections that do not represent such moment.

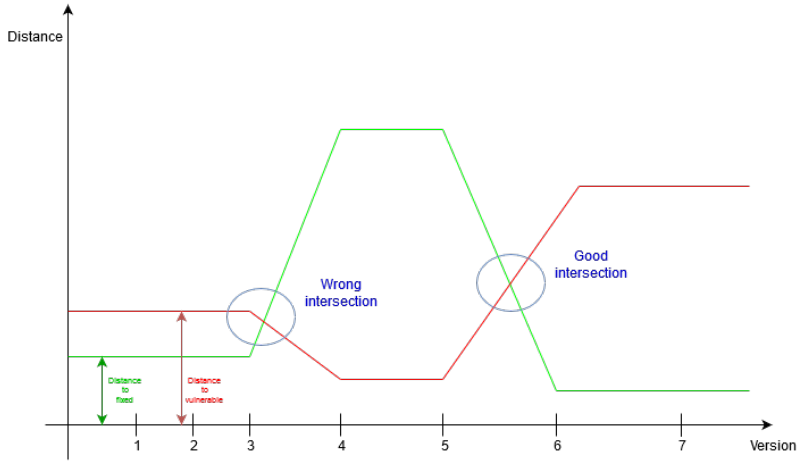


Figure 3.10: Example of good and wrong intersections.

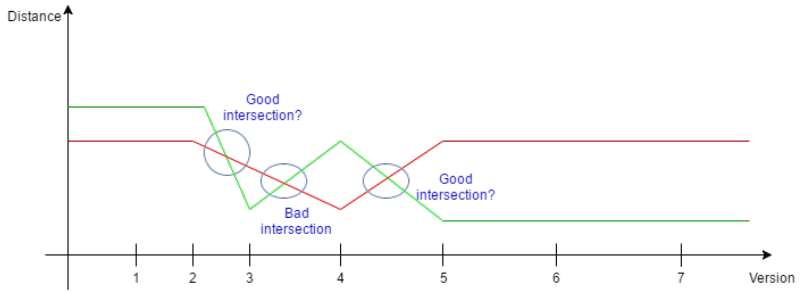


Figure 3.11: Example of many intersections.

first fixed is available, it allows to take as good only the intersection situated after last vulnerable and before first fixed. If still more than one intersection is found, they have to be discarded since non reliable conclusion can be taken, thus a human review is required.

When information about last vulnerable and first fixed is not available, only the check that the intersection within the same major release is unique can be applied.

After having found and validated all the intersection, it is possible to perform the assessment for different versions according to their respective major release:

$$if\ version \leq intersection\ start \Rightarrow version\ is\ vulnerable$$

$$if\ version \geq intersection\ end \Rightarrow version\ is\ fixed$$

After having validated the intersections, and used them in order to assess new archives, there might still be entire sets of archives belonging to the same major release for which no assessment can be done. In fact, if a construct is fixed in a certain major release, then slightly changed (but keeping it fixed) and a new major release is released, there will never be equality either to the vulnerable or fixed version, and at the same time no intersections will be found on that major release, even though all the versions belonging to it are fixed. In order to cope with this case, a new assumption has to be done:

Assumption 4. *If, for the library under analysis and for a certain major release:*

- *None of the versions belonging to the major release can be assessed thanks to the equality of the contained constructs,*
- *No intersection has been found on the major release,*

- *One intersection has been found for another major release and the version corresponding to intersection end has been released before the first version belonging to the major release,*

then all the versions belonging to the major release are fixed.

Is it possible to assess, in a similar way, a major release whose last version has been released before the fix of the vulnerability?[7] The same reasoning that led to Assumption 4 cannot be applied, since major release might either be:

1. Fixed, since the vulnerability might not have been introduced yet;
2. Vulnerable, since the vulnerability might exist but has not been fixed yet.

Since there is not enough information that allows for discriminating between these two cases, no reliable assessment can be performed.

3.5 Implementation

In this section I will explain some details about the implementation for the approach described in previous sections. The approaches have been implemented incrementally, using the first one (described in section 3.2) as a starting point and then including it in the new ones in order to be able to assess a bigger number of archives and, at the same time, obtain more reliable assessments.

3.5.1 Assessment by equality

The starting point for the approach is the assessment of the single constructs belonging to the change list for the bug under analysis. The change list, just like in version check, can be retrieved from vulas backend. As a first approach, only constructs that have been modified as part of the fix are analyzed, because they are the only one for which both vulnerable and fixed version are available.[8]

The fact that a construct might have been modified in multiple commits must be taken into account: this is done by sorting all the commits where the construct has been modified, considering only the first and the last one. Differently from version check it is no longer interesting to look at the changes necessary for going from vulnerable to fixed version, but only the initial vulnerable and last fixed source code are needed. These are in fact the only versions of the source code that might be found when analyzing the released archives. The 'might be found' is justified by the fact that the vulas backend only stores the commits that are security related (respect to the bug under analysis), then it might happen that code equality cannot be found.

Having the vulnerable and fixed code for all the modified constructs of the change list allows to analyze the archive and look, for each of them, for the equality of the code. The source code for the archive must, of course, be available in order to perform this step. In order to be compared, the source code needs to be represented in a consumable way and, just like in version check, Abstract Syntax Trees are used and then compared by Change Distiller. Change Distiller produces the edit scripts containing the changes necessary to go from one AST to another: the number of this changes can represent the distance between the ASTs.

If the construct is found in the archive, it can exist in one of the following states:

[7] Similarly meaning that no version belonging to the major release under analysis has been assessed, no intersection has been found on the major release and the last version of the major release has been released before the version representing the intersection start of an intersection found on another major release.

[8] For constructs that have been added as part of the fix, only fixed code is available. On the other hand, for deleted constructs, only vulnerable code is available.

- **Vulnerable** : The code of the construct found in the archive is exactly equal to its vulnerable version. This is verified by computing the edit script vulnerable to test and checking that it is empty, namely the number of changes is zero;
- **Fixed** : The code of the construct found in the archive is exactly equal to its fixed version. This is verified by computing the edit script test to fixed and checking that it is empty, namely the number of changes is zero;
- **Not assessable** : The code of the construct found in the archive is neither exactly equal to its vulnerable nor to its fixed version. This is verified by computing both the edit script test to fixed and the edit script vulnerable to test, and checking that both of them are not empty, namely in both cases the number of changes is different from zero.

As a security check for the approach, it is worthy verifying that each construct is not both vulnerable and fixed at the same time, case in which there would be inconsistency. Once the assessment is obtained for all the constructs, and no inconsistency has been found, it is possible to establish the state of the archive:

- **Vulnerable** : at least one construct has been found to be equal to its vulnerable version;
- **Fixed** : at least one construct has been found to be equal to its fixed version;
- **Unknown** : the archive does not meet the requirements for being classified neither as fixed nor as vulnerable.

At the time of writing, only modified constructs are taken into account, therefore the archive might still contain constructs that have to be deleted or might not yet contain constructs that have to be added as part of the fix. An archive might be in the unknown state because all the constructs, but one, contained in it are fixed: this is one shortcoming of this algorithm that can be overcome by exploiting the information coming from the assessment for all the archives sharing the same group and artifact.

3.5.2 Improvement with major release

In order to improve the approach described in 3.5.1, the information coming from the assessment of all the archives sharing the same group and artifact can be exploited.

The approach consists in:

1. Computing the assessment individually for each archive;
2. Sorting the archives by time-stamp of release;
3. For each construct, finding its last vulnerable and first fixed versions (if any);
4. Aggregate the information coming from all the constructs and all the archives in order to find the last vulnerable and first fixed version for the archives.

By using this approach, it could be possible to assess all the archives released before the last vulnerable as vulnerable and all the ones released after the first fixed as fixed. Unfortunately, this would lead to wrong assessments for versions not affected by the bug, but released before the bug was inserted in the code.[9] As it is, this improvement cannot bring new assessments since all the versions for which an assessment can be done are only the ones having equality to vulnerable or fixed for at least one of their constructs, thus the ones assessed by using the approach in 3.5.1.

In order to exploit the assessment of multiple archives, it is necessary to divide the set of releases by their relative major release. The approach consists in:

[9] Discussion about this problem can be found in the example shown in section 3.3.

1. Computing the assessment individually for each archive;
2. Dividing the archives by major release;
3. Sorting the archives by time stamp of release;
4. For each construct, finding its last vulnerable and first fixed versions (if any);
5. Aggregating the information coming from all the constructs and all the archives (within the same major release) in order to find the last vulnerable and first fixed version (within the same major release);
6. Performing the assessment according to the last vulnerable and first fixed version found within the same major release.

The assessment for each archive can be:

$$archive\ a\ is\ vulnerable \iff a \in major\ release\ MR \wedge \exists\ last\ vulnerable\ LV \in MR \mid a \leq LV$$

$$archive\ a\ is\ fixed \iff a \in major\ release\ MR \wedge \exists\ first\ fixed\ FF \in MR \mid a \geq FF$$

The division per major releases allows to assess more libraries and to be more precise in the assessment. However, since it still tightly tied to assessment by equality, it does not allow to assess the cases for which either last vulnerable or first fixed (or both) are not found.

3.5.3 Assessment by distance

In order to cover the assessment of a larger number of libraries and to find the moment when the fix has been introduced, it is possible to study the behavior of the distances of all the constructs from the change list from their respective vulnerable and fixed version. Starting from the approach described in section 3.5.1, the distance can be computed as the number of changes contained in the respective edit script. Therefore distance to vulnerable is the number of changes contained in the edit script vulnerable to fixed, whereas distance to fixed is the number of changes contained in the edit script test to fixed. The same evaluations that brought to the division by major release, and the approach implemented in section 3.5.2, still holds and must be taken into account.

In order to find the intersections by using the distances, the difference between distance to fixed and distance to vulnerable must be computed. Having this difference for all the versions where the construct is contained, allows to compare the difference among two subsequent releases, finding the intersection if there is a switch in the sign of the difference. After having found an intersection, it needs to be validated in order to check if it has the good direction, and this is obtained by checking that the difference between the two distances is positive before the intersection, and negative afterwards. This is shown in Figure 3.12 and relative Listing 3.1.

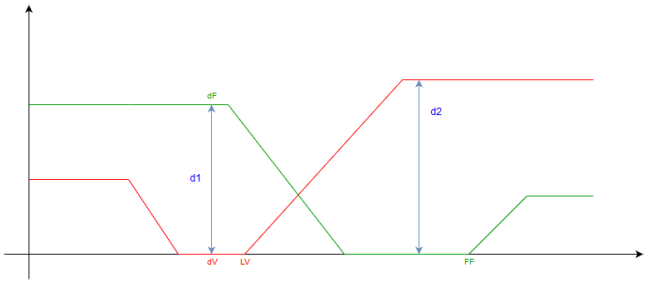


Figure 3.12: Representation of how intersection is found.

```
dV1 = vulnerableToTest.size()
dF1 = testToFixed.size()
difference1 = dF1 - dV1;

dV2 = vulnerableToTest.size()
dF2 = testToFixed.size()
difference2 = dF2 - dV2;

if (difference1 >=0 && difference2 <0) {
// goood intersection between version 1
     and version 2
}
```

Listing 3.1: Sample source code

The intersections need to be found per major release, the approach that does it consists in:

1. Computing the distances for each construct in each archive;
2. Dividing the archives by major release;
3. Sorting the archives by time-stamp of release;
4. For each construct, looking for the intersection (if any);
5. Aggregating the information coming from all the constructs and all the archives (within the same major release) in order to validate the found intersections (within the same major release).

In order to validate the intersections found within each of the major release, there are two possible checks:

- If the information about last vulnerable is available, the intersection is valid only if it happens after last vulnerable and, similarly, if the information about first fixed is available, the intersection is valid only if it happens before first fixed;
- Verify that the intersection is unique within the major release.

These checks are performed independently because, in case information about last vulnerable and first fixed is not found, it might still be possible to perform the assessment if the intersection is unique within the major release. The validation of intersections ensures that any possible noise coming from the creation and comparison of edit scripts is not affecting the results of the assessments.

Once all the intersections have been found and validated, it is possible to use them in order to assess the archives:

$$Let\ IS\ be\ intersection\ start\ for\ intersection\ I$$

$$Let\ IE\ be\ intersection\ end\ for\ intersection\ I$$

$$archive\ a\ is\ vulnerable \iff a \in major\ release\ MR \land \exists\ intersection\ I \in MR \mid a \leq IS$$

$$archive\ a\ is\ fixed \iff a \in major\ release\ MR \land \exists\ first\ fixed\ FF \in MR \mid a \geq IE$$

After these assessments, it is possible to check if entire sets of major releases have not been assessed and mark them as fixed, as described for Assumption 4:

$$Let\ IE\ be\ intersection\ end\ for\ intersection\ i$$

$$Let\ first(Mr)\ be\ the\ first\ released\ archive\ for\ Mr$$

$$If:$$

$$\forall\ archive\ a \in Major\ Release\ Mr : \nexists\ a \mid a\ is\ vulnerable \lor a\ is\ fixed \land$$

$$\nexists\ intersection\ i \mid i \in Mr \land \exists\ intersection\ i \mid i \notin Mr \land first(Mr) < IE$$

$$\Rightarrow mr\ is\ fixed$$

3.5.4 Final algorithm

In order to perform the assessment of the biggest possible number of archives and, at the same time, obtain reliable assessments that do not require manual review, thus overcoming the limitations of version check, I aggregated the previously described approaches into a single algorithm. The algorithm will be capable of assessing, at the same time, a large number of archives potentially affected by a certain bug. It is divided in three main phases:

1. Given a bug, the list of potentially affected libraries is retrieved from vulas backend and, by looking on Maven Central [18], all the versions of the potentially affected libraries sharing the same group and artifact identifier are retrieved. For each of the retrieved archives, approach described in section 3.5.1 is used in order to compute, for each modified construct found in the change list of the bug, the distance to its vulnerable and fixed version. In this phase, a first assessment is possible for the archives having the constructs equal to their vulnerable or fixed versions, as described in 3.5.1.

2. After having analyzed all the single archives, it is possible to combine the obtained information in order to find the last vulnerable and first fixed version of the archive for the bug and, if more than one major release is available, find the last vulnerable and first fixed version within the major releases. This information is used in order to assess the archives as described in section 3.5.2.

3. Having already assessed a certain amount of archives in the first two phases, only the approach using the intersections, as described in section 3.5.3, is left. If many archives have already been assessed in the first two phases, in this phase few assessment will be performed. However, since valid intersections represent the moment when the fix for the bug has been released, it is an interesting information to be found.

The three phases of the algorithm are shown in Figure 3.13.

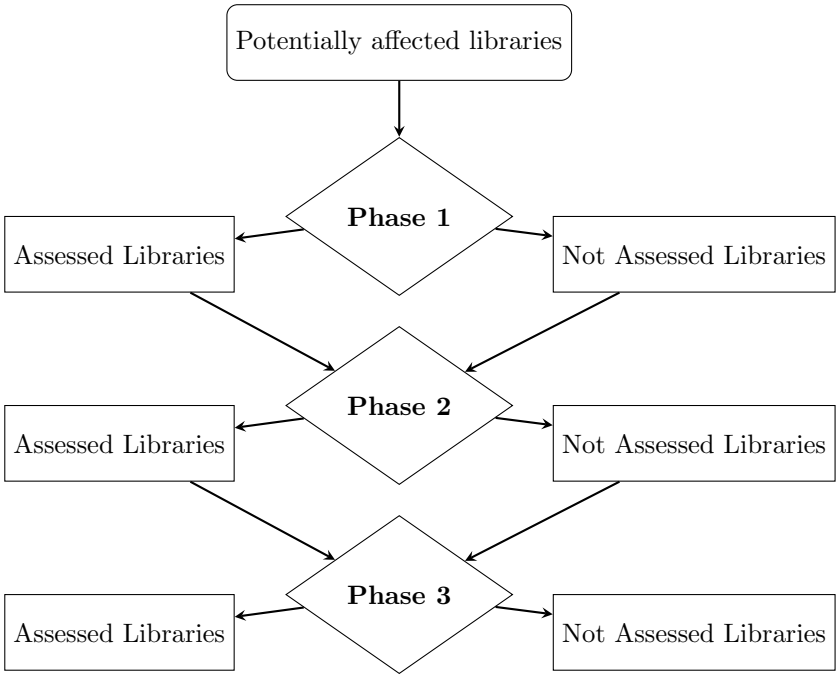


Figure 3.13: Three phases of the algorithm.

I will describe the evaluation of the algorithm and the results obtained in chapter 4. The pseudo code for the algorithm can be found in Appendix A.3.

Chapter 4

Evaluation

In this chapter I will comment on the results obtained with the approach patch evaluator described in Chapter 3. I will first show the numbers of the archives that have been assessed with the patch evaluator approach, then I will show and comment some charts in order to illustrate the approach and, at the end of the chapter, I will discuss the evaluation of the results.

4.1 Main results

In order to test and validate the new approach, I have used all the bugs known by vulas backend (115 at the time of writing). For each bug, different archives sharing the same group and artifact might be affected: in my analysis, I have found 7405 archives potentially affected, 6620 of them having sources available (thus 6620 cases in which the approach can be applied).

- Phase 1 produces 3515 reliable assessments, namely 53% of the total of the archives with sources.
- Phase 2 produces 841 new reliable assessments, bringing the total number of assessments up to 4356, namely 66% of the total of the archives with sources.
- Phase 3 produces 352 new archives using intersections, 986 by using the information about not affected major releases, bringing the total number of assessments up to 5694, namely 86% of the total of the archives with sources.

By analyzing the results obtained per construct and archive, 29810 distinct couples (construct,archive) are found, and, among them, in 11223 cases there is equality between the construct found in the archive and either its vulnerable or fixed version (roughly 38% of the cases). The stability of the code of the constructs as released in different archives comes as confirmation of the metrics computed in Section 2.5.

4.2 Illustrative use cases

In this section I will present some charts that offer a graphical representation of the approach. The different releases for the archive under analysis are sorted by chronological order on the X-axis of the charts, the distances of the construct under analysis (as found in the archive under analysis) respect to its vulnerable and fixed versions are on the Y-axis.

CVE-2012-2098

NVD description: *Algorithmic complexity vulnerability in the sorting algorithms in bzip2 compressing stream (BZip2CompressorOutputStream) in Apache Commons Compress before 1.4.1 allows remote attackers to cause a denial of service (CPU consumption) via a file with many repeating inputs.*

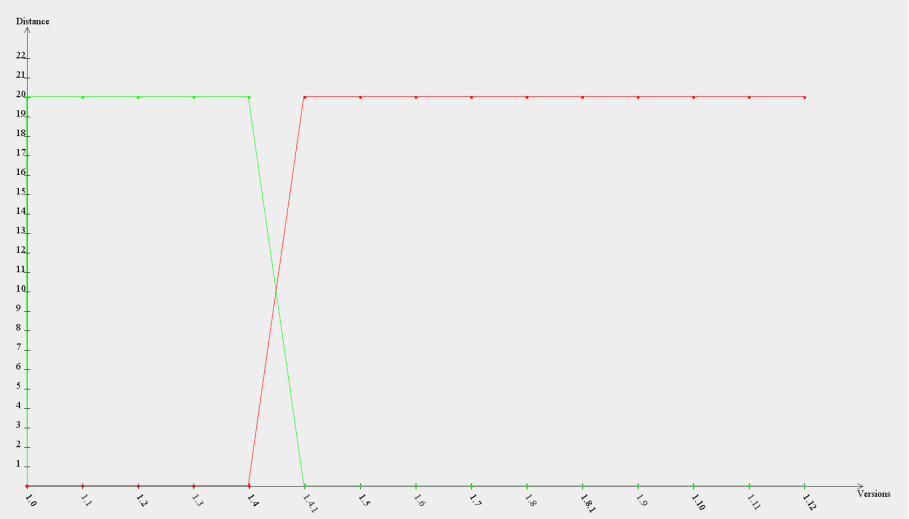


Figure 4.1: CVE-2012-2098 - Distances for method *blockSort()*.

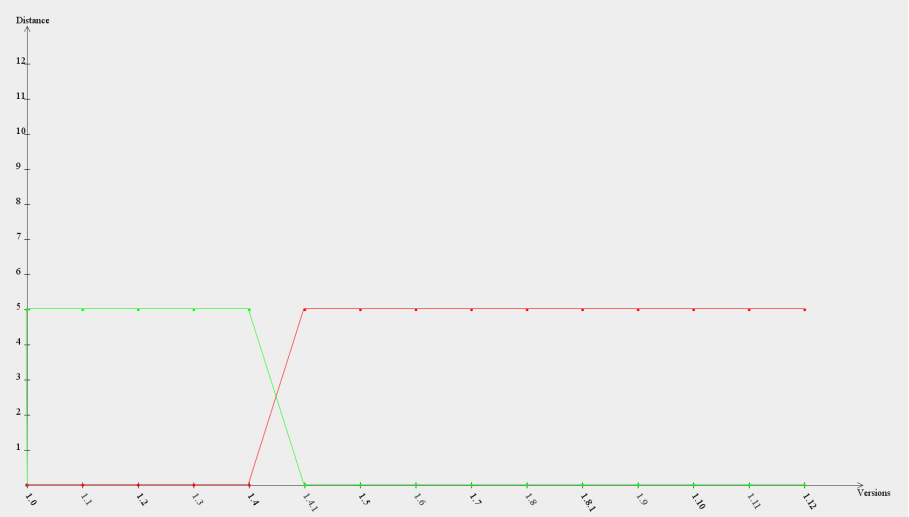


Figure 4.2: CVE-2012-2098 - Distances for method *endBlock()*.

As shown in Figures 4.1 and 4.2, for two different constructs *blockSort()* and *endBlock()*, equality to respective fixed and vulnerable version is found for all the archives. In this case all the archives can be assessed during the phase 1 of the algorithm and, by looking at the intersection found between versions *1.4* and *1.4.1*, it is possible to notice how it reflects NVD's description for this bug.

CVE-2015-5345

NVD description: *The Mapper component in Apache Tomcat 6.x before 6.0.45, 7.x before 7.0.68, 8.x before 8.0.30, and 9.x before 9.0.0.M2 processes redirects before considering security constraints*

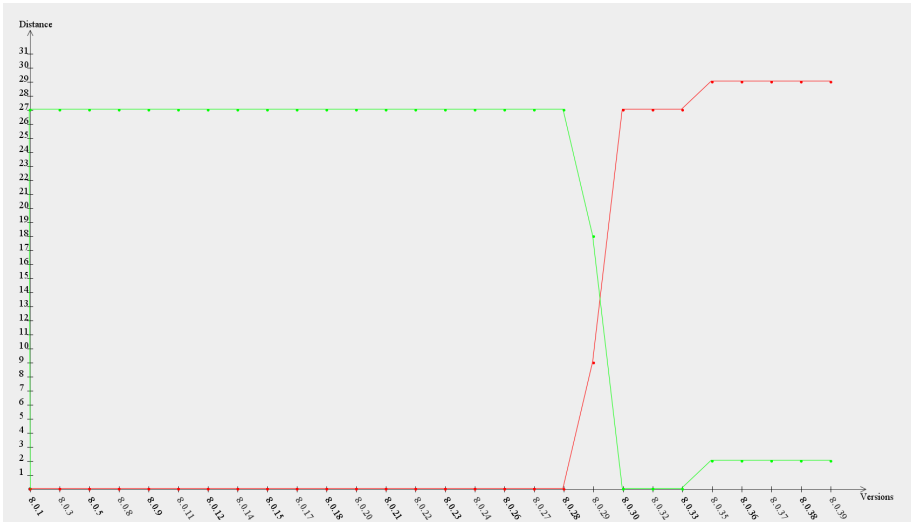


Figure 4.3: CVE-2015-5345 - Distances for method *internalMapWrapper()* and major release 8.

and Filters, which allows remote attackers to determine the existence of a directory via a URL that lacks a trailing / (slash) character.

As shown in Figure 4.3, the assessment for versions belonging to major release 8 has been possible thanks to division by major release, during phase 2 of the algorithm. In fact, having found version *8.0.30* as fixed allows to assess all the subsequent releases (within major release 8) as fixed. The obtained results reflect NVD's description for this bug.

CVE-2014-0050

NVD description: *MultipartStream.java in Apache Commons FileUpload before 1.3.1, as used in Apache Tomcat, JBoss Web, and other products, allows remote attackers to cause a denial of service (infinite loop and CPU consumption) via a crafted Content-Type header that bypasses a loop's intended exit conditions.*

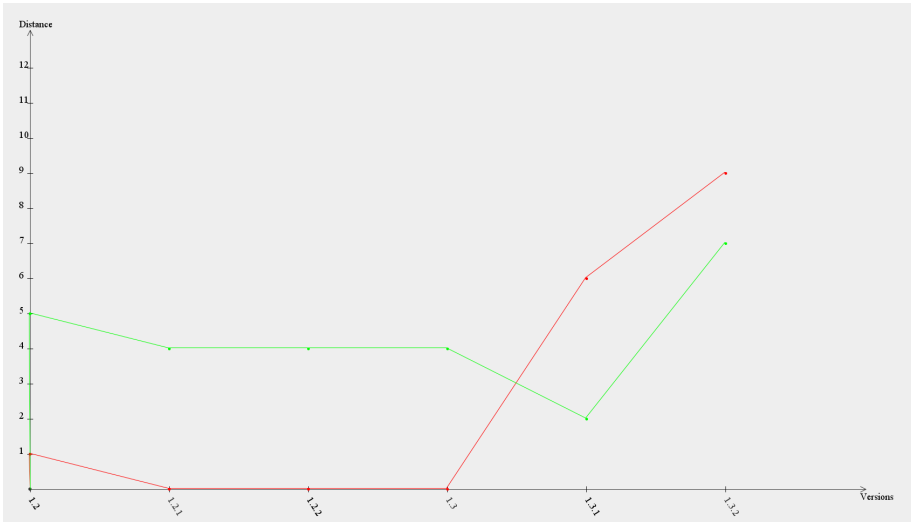
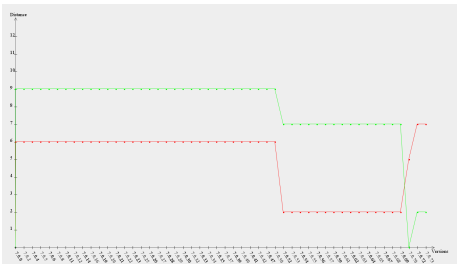


Figure 4.4: CVE-2014-0050 - Distances for method *MultiPartStream()*.

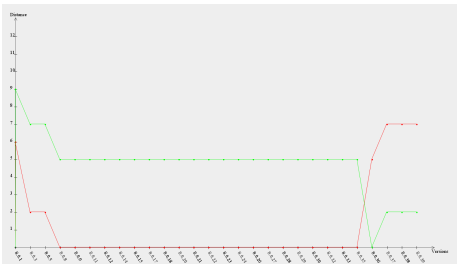
As shown in Figure 4.4, the assessment of versions *1.3.1* and *1.3.2* is possible thanks to the intersection found between versions *1.3* and *1.3.1*, even though no version having equality with the fixed code has been found. The assessment of the fixed version happens during phase 3 of the algorithm, and the obtained results reflect NVD's description for this bug.

CVE-2016-3092

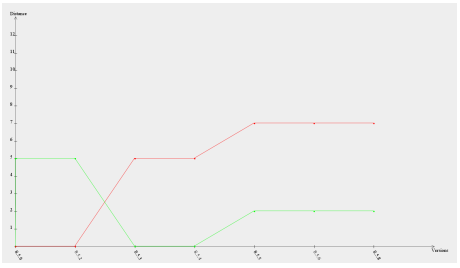
NVD description: *The MultipartStream class in Apache Commons Fileupload before 1.3.2, as used in Apache Tomcat 7.x before 7.0.70, 8.x before 8.0.36, 8.5.x before 8.5.3, and 9.x before 9.0.0.M7, allows remote attackers to cause a denial of service (CPU consumption) via a long boundary string.*



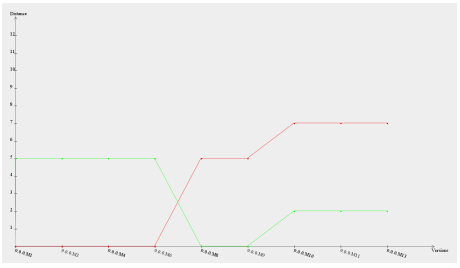
(a) Major release 7.



(b) Major release 8.



(c) Major release 8.5.



(d) Major release 9.

Figure 4.5: CVE-2016-3092 - Distances for method *MultiPartStream()* in different major releases.

As shown in Figure 4.5, the approach is capable of finding in different major releases the moments when the fixes were introduced, in this case:

- Major release 7 : between versions 7.0.69 and 7.0.70
- Major release 8 : between versions 8.0.35 and 8.0.36
- Major release 8.5 : between versions 8.5.2 and 8.5.3
- Major release 9 : between versions 9.0.0.M6 and 9.0.0.M8

By comparing the previous results with NVD's description for this bug, it is possible to verify their correctness.

CVE-2014-3529

NVD description: *The OPC SAX setup in Apache POI before 3.10.1 allows remote attackers to read arbitrary files via an OpenXML file containing an XML external entity declaration in conjunction with an entity reference, related to an XML External Entity (XXE) issue.*

As shown in Figures 4.6 and 4.7, version *3.11-beta1* contains the fixed version of the construct *parseContentTypesFile()*, but also the vulnerable version of the construct *readFrom()*.

This case is useful in order to show how Assumption 3 is needed in order to perform reliable assessments. In fact, version *3.11-beta1* is only partially fixed:

- Construct *parseContentTypesFile()* is equal to its fixed version;
- Construct *readFrom()* is equal to its vulnerable version.

As stated by Assumption 3, since an archive having at least one construct equal to its vulnerable version is considered to be vulnerable, 3.11-beta1 needs to be assessed as vulnerable. The obtained results reflect NVD's description for this bug.

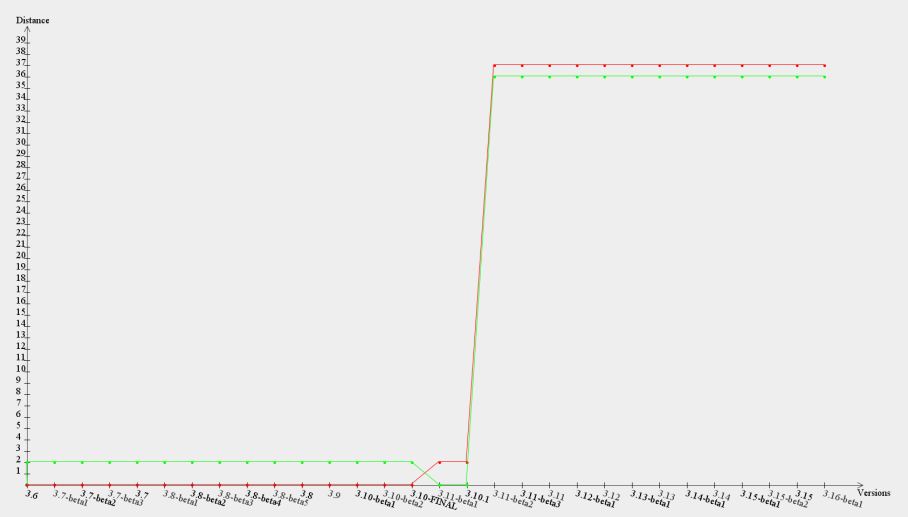


Figure 4.6: CVE-2014-3529 - Distances for method *parseContentTypesFile()*.

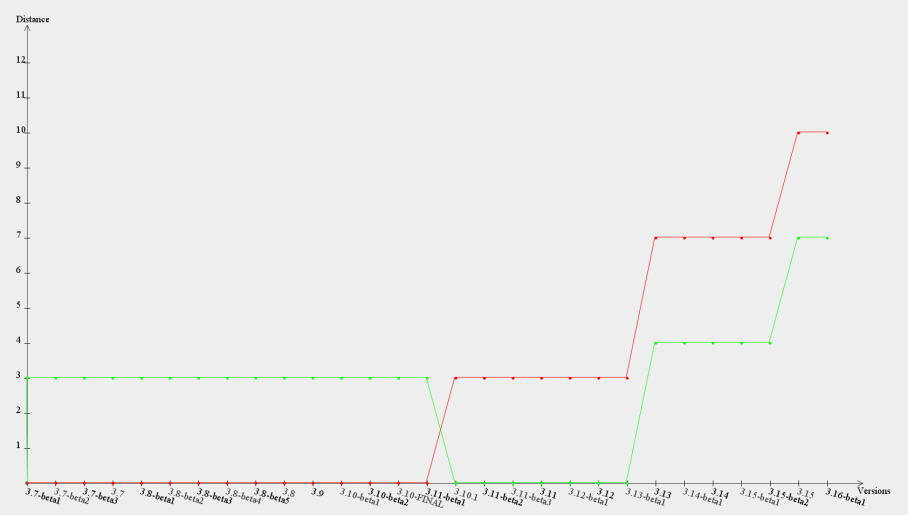


Figure 4.7: CVE-2014-3529 - Distances for method *readFrom()*.

CVE-2011-2526

NVD Description: *Apache Tomcat 5.5.x before 5.5.34, 6.x before 6.0.33, and 7.x before 7.0.19, when sendfile is enabled for the HTTP APR or HTTP NIO connector, does not validate certain request attributes, which allows local users to bypass intended file access restrictions or cause a denial of service (infinite loop or JVM crash) by leveraging an untrusted web application.*

As shown in Figure 4.8, an intersection is found for major release 7, whereas no intersection is found for major release 8, which also has no versions having equality with fixed code for any of the constructs. In this case it is possible to assess all versions belonging to major release 8 as fixed by using Assumption 4, since the fix has been released in version 7.0.19 that is released before 8.0.1, the first version belonging to major release 8. The obtained results reflect NVD's description for this bug.

CVE-2012-0213

NVD description: *The UnhandledDataStructure function in hwpf/model/UnhandledDataStructure.java in Apache POI 3.8 and earlier allows remote attackers to cause a denial of service*

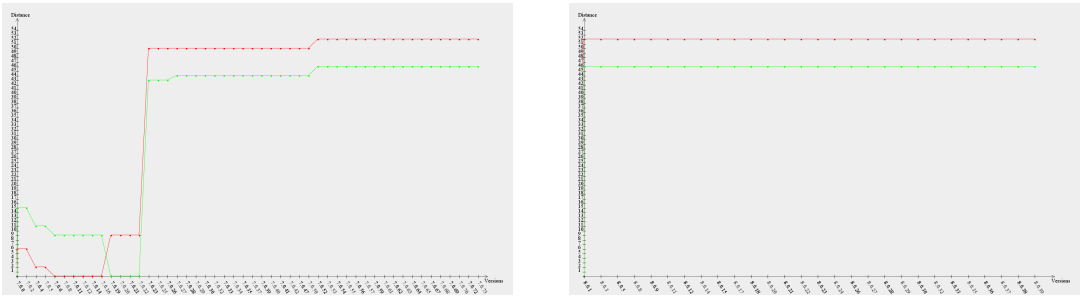


Figure 4.8: CVE-2011-2526 - Distances for method *setAttribute()* divided per major release. Major release 7 on the left side and major release 8 on the right side.

(OutOfMemoryError exception and possibly JVM destabilization) via a crafted length value in a Channel Definition Format (CDF) or Compound File Binary Format (CFBF) document.

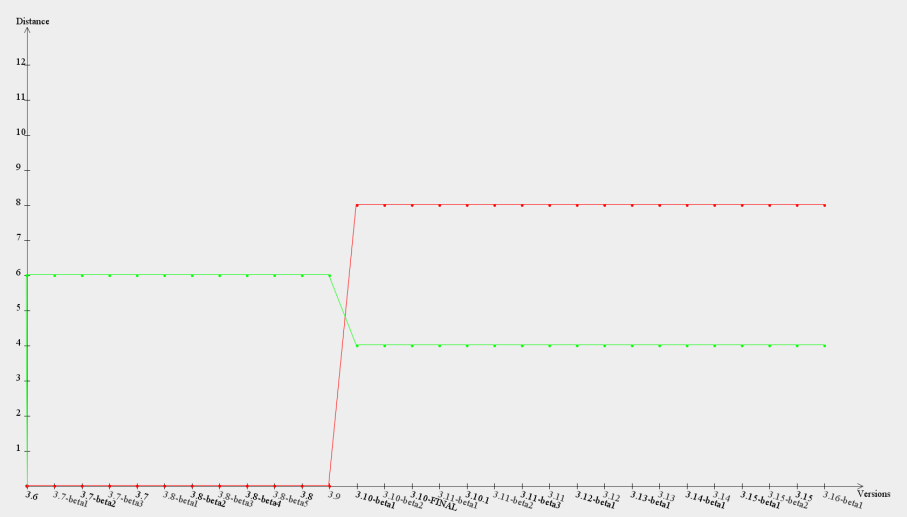


Figure 4.9: CVE-2012-0213 - Distances for method *UnhandledDataStructure()*.

According to NVD's description, this bug should have been fixed in version 3.8 of Apache POI, but the code for the method *UnhandledDataStructure()*, as visible from Figure 4.9, has been found exactly equal to its vulnerable version. The change list for this bug only contains this method, and being vulnerable means that the fix has been actually fixed only starting from version *3.10-beta1*, as indicated by the found intersection. In order to verify the correctness of the assessment for this bug, since the result was not compliant with NVD, I manually checked the sources for versions 3.8, 3.9 and 3.10-beta1. The code I found in versions 3.8 and 3.9 was equal to the vulnerable version, whereas it was different for 3.10-beat1, thus I could exclude any mistake during the comparisons of ASTs, and so verify the correctness of the assessment. Thanks to the found intersection, even though there are no constructs with equality to fixed, it is possible to assess all the versions after 3.10-beta1 as fixed.

CVE-2014-0095

NVD description: *java/org/apache/coyote/ajp/AbstractAjpProcessor.java in Apache Tomcat 8.x before 8.0.4 allows remote attackers to cause a denial of service (thread consumption) by using a "Content-Length: 0" AJP request to trigger a hang in request processing.*

In Figure 4.10 the distances computed for method *finish()* show how the division by major releases helps in better assessing the results and finding which versions are interested by the bug. If the division by major release was not performed, a lot of intersections would be found, thus

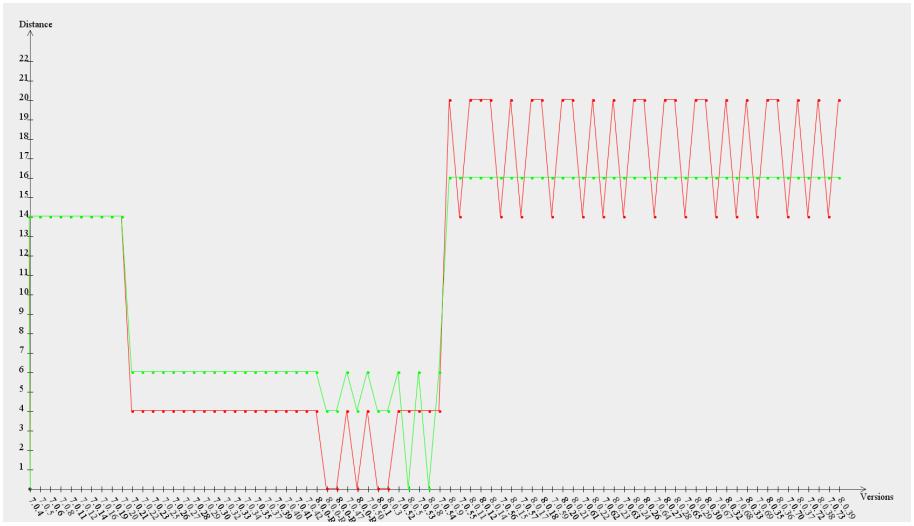


Figure 4.10: CVE-2014-0095 - Distances for method *finish()*.

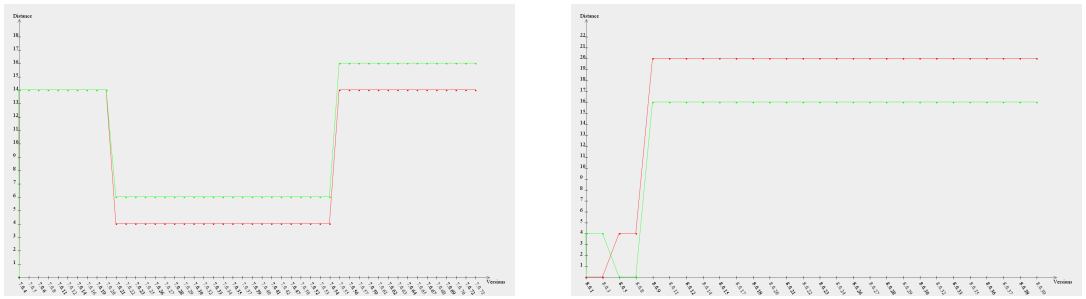


Figure 4.11: CVE-2014-0095 - Distances for method *finish()* divided per major release. Major release 7.0 on the left side and major release 8.0 on the right side.

none of them would be taken as good for performing any assessment[1]. As stated by NVD, only versions between *8.x* and *8.04* are affected by this bug, therefore only one intersection is expected to be found on major release 8. As shown in Figure 4.11, by dividing per major release, the only intersection is found between versions *8.0.3* and *8.0.5*, reflecting NVD's description for this bug.

For this CVE, the only assessed versions are the one having 8 as major release, whereas versions having 7 as major release are not assessed, since there is not enough information in order to guarantee that such versions are either fixed or vulnerable:

- For none of the constructs there is equality neither with the vulnerable nor the fixed version;
- No intersection has been found on the major release.

As described after Assumption 4 about the not affected major releases, in this case no assessment is done for versions belonging to major release 7, since there is not enough information that would allow to classify them either as vulnerable or fixed.

4.3 Evaluation

Given the relatively limited number of bugs available, I have been able to perform a quick check of correctness for all of them, using the following approach:

[1] As described in section 3.4.1.

- **Phase 1 and 2**: check for each major release the last vulnerable and first fixed version. If last vulnerable and first fixed are compliant with NVD's description, the results given in phase 1 and 2 of the algorithm are good.
- **Phase 3**: check if any intersection has been found and verify that it reflects NVD's description for the bug under analysis. In case the intersection(s) do not reflect NVD's description, manually verify by analyzing the source code of the constructs in the potentially misclassified archives, comparing it with their vulnerable and fixed version.

If for verification of both phases *1* and *2* I have never found any discrepancy, by verifying phase *3* I was able to identify some cases for which NVD's description of the bug was in conflict with the obtained results. Manual inspection revealed that the assessment given by patch evaluator was correct.[2].

At the time of writing, patch evaluator is not using all the information contained in the change list, but only the constructs that have been modified as part of the fix: vulas also offers the information about added and deleted constructs as part of the change list. In the next section I will describe how this can be used to further validate the results.

4.3.1 Evaluation through added or deleted constructs

In order to perform a better evaluation of the results obtained with patch evaluator, I implemented a new algorithm that exploits the information about the constructs that were added or deleted as part of the security fix for the bug under analysis.

Given the change list for a bug, and considering only added and deleted constructs, an archive is fixed if:

- It contains all the constructs that were added as part of the security fix;
- It does not contain any of the constructs that were deleted as part of the security fix.

This represents a partial assessment for the archive under analysis, that can be compared with the one given by patch evaluator for the same archive (and, of course, the same bug) in order to verify that no discrepancies exist between the two assessments.

When comparing the two assessments, I used an important constraint:

- In order to be assessed as vulnerable by only looking at added and deleted constructs, the archive must not belong to a major release that is not affected by the bug, otherwise it might be wrongly assessed. This has to be checked since, if the major release is not affected, it might not contain the constructs that are supposed to be added, fact that would assess it as vulnerable.

The total number of discrepancies emerged during the analysis is *8*. By analyzing such discrepancies, in all the cases:

- The archive is assessed as vulnerable by patch evaluator;
- The archive is assessed as fixed according to added and deleted constructs;
- By checking NVD's description for the bug and distances charts, the assessment given by patch evaluator resulted to be correct.

Since no other discrepancies has emerged, the evaluation performed through added and deleted constructs allows to trust the results given by patch evaluator.

[2]One example of this is shown in section 4.2.

4.3.2 Comparison with version check's results

By comparing the results obtained with version check (Table 2.2) with the ones obtained using patch evaluator (Table 4.1), the improvement is remarkable: by using the same set of CVEs and archives, no wrong assessment is obtained. Patch evaluator is in fact only performing an assessment when enough information to give the correct assessment are available, otherwise no decision is taken. For all the archives listed in Table 4.1, always the good assessment has been performed.

Bug	Assessment result Vulnerable version	Assessment result Fixed version	Assessment result Latest version
CVE-2009-2625			
CVE-2011-1498			
CVE-2012-0838			
CVE-2012-2098			
CVE-2012-6153			
CVE-2013-2186			
CVE-2014-0050			
CVE-2014-3529			
CVE-2014-3574			
CVE-2014-3577			
CVE-2016-2162			
CVE-2016-3081			
CVE-2016-3082			

Table 4.1: Correctness of results obtained. (X indicates wrong assessment)

Even though both the approaches are based on source code comparison, they are very different: version check tries to find the single changes that happened in any of the constructs of the change list, whereas patch evaluator has as its basis the equality of the whole source code of the constructs. As shown, the results given by patch evaluator are much more reliable and provide more information (when an intersection is found, it shows when the bug has been fixed), then some solutions must be studied in order to use patch evaluator in place of version check. At the time of writing patch evaluator cannot be integrated into vulas since, like for version check, in order to analyze a library found in the application under analysis it is necessary to decompile the code of its constructs, and the original source code is not available. Some possible implementation of the integration will be discussed in Section 4.4.1.

4.3.3 Limitations

In this section I will describe the main limitations of the patch evaluator approach.

Missing Sources

Patch evaluator is actually taking into account only archives having sources available, since they are the only cases for which it is possible to compare the source code for the constructs. The number of archives (not distinct) without sources is 785, thus an important number of archives that are not taken into account, and therefore not assessed.

The category of repackaged libraries falls among the libraries for which sources are not available. A repackaged library is a library that has been modified, in some of its components, by a third-party that is not the original library developer. In such cases the distributed sources are only containing the classes that have been modified by the third-party, whereas the whole library is contained in the packaged version: this makes the packaged version potentially affected by the bugs affecting the original library, but it does not allow anymore to find and match the fixed (vulnerable, resp.) source code.

Constructs Added or Deleted not taken into account for the assessment

The current implementation is only considering the constructs that have been modified as part of the fix for the bug in order to perform the assessments. However, in the change list for a certain bug, there are constructs that have been either added or deleted as well, and they have only been used as validation of the results, as described in Section 4.3.1. Added and deleted constructs are a source of potential very useful information since[3]:

- If an archive does not contain a construct that should be added as part of the fix, the archive is vulnerable;
- If an archive contains a construct that should be deleted as part of the fix, the archive is vulnerable.

For the bugs that have been studied, this limitation has not lead to wrong assessments, but it will need to be taken into account.

4.4 Future directions

Given the limitations listed in Section 4.3.3, I will now describe some ideas in order to improve patch evaluator.

Use the information about added and deleted constructs

Given the change list for a certain bug, it contains constructs that have been modified, added or deleted as part of the fix. At the time of writing, patch evaluator only uses the information about the modified constructs in order to perform the assessments, since they are the only ones for which it is possible to compare the version of the code before and the one after the fix. Added and deleted constructs can, potentially, be used in order to assess archives for which sources are not available since, even though source code is not available, it is still possible to look whether or not the archive contains the constructs under analysis.

Decompile the packaged libraries

For all the archives that have not the sources available for download, the decompilation of the contained constructs is the only existing possibility in order to perform the assessment. Decompilation has, however, different shortcomings[4] and it would not allow to look for exact equality of the decompiled code with the original source code for the same construct (since it is unlikely that, for a certain construct, the source code and the decompiled code exaclty match). In order to overcome problems linked to decompilation, a mixed approach could be used for each construct of the change list:

1. Find, with the existing approach, the versions of the archive (for which sources are available) containing the source code of the construct in its vulnerable and fixed state;
2. Get the packaged version of the archives having equality;
3. Decompile the code of the construct within the packaged version of the archives having equality and use it for comparison with the decompiled code of the construct as found in the archive without sources (both for vulnerable and fixed version) instead of using the original source code.

By using the decompiled code of the vulnerable and fixed version of each construct, the very same algorithm can be used.

[3] Further details about the change list are available in Section 2.1.1.

[4] Some of the shortcomings related to decompilation are listed in Section 2.2.2.

Discover when the vulnerability was inserted

For a certain construct, as discussed in Section 3.4, it is theoretically possible to find out, by analyzing the behavior of the distance of the code under analysis with its vulnerable and fixed versions, when the bug has been fixed. Similarly it might be possible, by analyzing the same distances, to find out when the vulnerability has been inserted in the code of the construct under analysis. The intersections marked as wrong (see Section 3.4.1) could represent possible candidates of versions of the archive when the bug was first inserted.

Change source code representation and comparison

Patch evaluator approach is based on source code comparison through AST representation and their comparison through change distiller. Although obtained results are good, allowing to assess 86% of the analyzed libraries (having sources available), these representations and comparisons do not allow the algorithm to take any decision when neither equality nor intersections have been found. If different solutions were studied and implemented in order to represent the distances to vulnerable and fixed version of a certain construct, it could be possible to perform assessments not looking for exact equality, but simply for smaller distances. Possible different source code representation might be Control Flow Graphs [19] or Program Dependence Graphs [20].

4.4.1 Integration with vulas

Having demonstrated that patch evaluator produces more reliable results than the ones produced by version check, it is necessary to think of how to integrate it in vulas in order to replace version check.

The version check approach relies on the decompilation of the library found within the application under analysis, in order to give an immediate feedback about the vulnerability state of the library and thus the application including it. With patch evaluator this is no longer possible, since it aims at assessing at once all the libraries potentially affected by a certain bug: it is therefore necessary to find a link between the libraries assessed by patch evaluator and the library found within the application under analysis.

Three different approaches can be considered:

1. For each library, its group, artifact and version are known. Having this triplet allows to identify an unique archive on Maven central, for which both sources (the starting point for the assessment) and packaged version are available (if the assessment for the library has been done by patch evaluator). By computing the SHA-1[21] of the library found within the application under analysis with the SHA-1 computed for the libraries analyzed by patch evaluator, it is possible to find an unique correspondence and thus assess the application's vulnerability state.

2. If it is not possible to identify the corresponding library with sources by using the SHA-1, it might still be possible to identify it by comparing the code of the libraries:

   (a) For each library with sources, get the respective packaged version and decompile the constructs belonging to the change list of the bug, obtaining their AST representation;

   (b) Decompile the constructs found in the library under analysis and, after having obtained their AST representation, compare it with the corresponding ones as found in (a);

   (c) If a match is found for all the constructs coming from the two libraries, then the two libraries are the same (or at least they are equivalent) and the assessment will be the same.

3. If it is not possible to find a correspondence by using the SHA-1 of the libraries, the patch evaluator approach must be extended to the decompiled code (as described in Section 4.4). Particularly, these steps might be followed:

(a) Run patch evaluator for the bug that potentially affects the library found in the application under analysis;

(b) Find a version of the library having exact equality with fixed code and another version having exact equality with vulnerable code (for each construct of the change list);

(c) Get the packaged version of the libraries for which equality has been found, decompile them in order to obtain the AST representation of decompiled code;

(d) Get the AST for the decompiled constructs of the library found in the application under analysis;

(e) Compare all the ASTs from the library found in the application under analysis with their vulnerable and fixed version, looking for equality.

If equality is found for all the constructs with a certain library, then assessment for the library under analysis will simply be the same given by patch evaluator for the matching library (like in point 2).

Chapter 5

Conclusions

The use of open source libraries has increased over the years, with many commercial products exploiting all the advantages offered by open source software. Given this large and larger use, it becomes very important to find and fix security bugs affecting open source libraries, since every vulnerability they contain might easily affect the applications including the library as well. Therefore, whenever a bug affecting a certain open source library is disclosed, it is important to understand whether the version of such library used by a certain application is affected by the vulnerability or not. It is not easy to automatically perform this task: in this thesis I have discussed two different approaches that try to solve this problem.

In the first approach, version check, I have studied the existing solution, trying to understand its main problems and limitations and, later on, how to overcome them. As discussed in Section 2.4, version check has been proved to have several limitations (mostly due to decompilation issues) that do not allow to trust its results. In fact, being based on the changes computed on the constructs in order to fix the vulnerability, it is subject to wrong assessments when the constructs are modified for other reasons (i.e. refactoring, fix of new bugs): the AST representation of the constructs, even if only slightly modified, does not allow anymore to match the original changes. This problem is particularly critical when comparing the source code of the constructs, as committed in the VCS where the library is developed, with the respective decompiled version as found in the library, with several decompiler/compiler optimizations that do not allow to match the source code. When it is no longer possible to find the changes within the source code under analysis, the library is assessed as vulnerable even if it is fixed, since the logic of the fix has been moved elsewhere or the version belongs to a major release that is not affected at all by the bug. I applied several improvements to the existing algorithm (as discussed in Section 2.3) decreasing, for the set of analyzed bugs, the number of wrong assessments from 46% down to 31%. By analyzing the results relative to the wrong assessments, it was easy to notice how the wrong assessment was often happening for recent versions of the library under analysis: it was then important to study some measure of closeness between the version of the library found in the application and the version when the fix has first been introduced. In order to measure such closeness I studied some quantitative metrics (in Section 2.5), coming to the conclusion that they cannot be used for the purpose.

Even though the computed metrics were not useful for establishing whether version check can be applied, they showed that classes and methods are quite stable over time. The confirmation for this observation has been drawn, as shown in Section 4.1, by analyzing the number of constructs having the source code exactly equal either to their vulnerable or fixed version, with 11223 distinct (construct,artifact) pairs having equality over the total of 29810 (construct,artifact) pairs analyzed. This stability has allowed for the creation of a completely new approach (patch evaluator), that was supposed to overcome the limitations of version check and also be used in order to assess a larger number of libraries.

The base of patch evaluator is the equality of source code (through AST comparison), looking among all the versions of a certain library for the version containing exactly the same source code, either vulnerable or fixed, for the constructs that have been modified as part of the fix for the bug under analysis. It is possible to assess, with the constraints introduced by Assumption 3, 53%

of the archives potentially affected by the bug and for which sources are available. Later on, the approach has been improved in order to exploit the information coming from the assessments for all the versions of the archive under analysis, making it possible to assess new versions just basing on the major release they belong to. Finally, the latest assessments are performed by exploiting the information about the distance of all the constructs contained in the library under analysis from their fixed (vulnerable, resp.) versions. In fact, by analyzing the evolution of the distances for the different constructs in the different versions of the library, it is possible to discover when the bug has been fixed and, consequently, establish the assessment for the different versions. Finally, the approach is capable of assessing 77% of the potentially affected archives, 86% if considering only the archives for which sources are available. As shown in Section 4.3.2, the improvement of the assessments' quality between version check and patch evaluator is remarkable, with no wrong assessment being performed by patch evaluator on the same set of bugs for which version check was tested.

Patch evaluator can be, of course, improved in order to cover the remaining percentage of archives for which no assessment has been possible. Most of the improvements must be done for the archives that have not downloadable sources and that therefore are not taken into account by the current algorithm: the information about the added/deleted constructs could be used for this purpose, but also the decompilation can be taken into account in order to cover 100% of the libraries[1].

[1] A wider discussion about the future directions can be found in Section 4.4

Appendix A

Pseudocodes

A.1 Signature Analysis

```
Name Convention
{
   FixContainmentCheckInput: vulnerability or archive provided by vulas backend
   pathToSrc: path containing /src/
   ArchiveFixContainmentCheck: class containing methods to decide if an archive contains or not a
        fix
   ConstructFixContainmentCheck: class containing methods to decide if a construct is fixed or not
   OrderedCommitsForPath: class containing a series if construct changes for a given construct,
        whereby all the changes concern the same software repository path. It also offers a
        couple of convenience methods to understand the oerall change introduced by the single
        changes.
   ASTUtil: class implementing different methods for comparing nodes
}

SignatureAnalysis.executeVulas():
//Get all the constructs to be checked from the vulas backend
vulasBackEnd.getVulnerableAppConstructs()
for each FixContainmentCheckInput :
   if notToBeChecked:
      continue
   end if
   //Get all construct changes
   FixContainmentCheckInput.getChangesOfContainedConstructs()
   //Get set of changes for distinct path fragments, checking whether every path contains /src/
   ConstructChange.getPathsToSrc()
   //Check fix containment for each path: if at least one path is fixed, archive is fixed
   archiveIsFixed = false
   for each pathToSrc :
      //Check if the archive is fixed
      archiveIsFixed ||= ArchiveFixContainmentCheck.containsFix()
   end for
end for

ArchiveFixContainmentCheck.containsFix():
//An archive is considered fixed if all constructs affected by a fix are contained in the archive
   archiveIsFixed = true
   for each construct:
      //Check if the construct is fixed
      constructFixed = ArchiveFixContainmentCheck$ConstructFixContainmentCheck.containsFix()
      //update the archive is fixed information by anding archiveIsFixed with the result about the
           construct
      archiveIsFixed &= constructFixed
   end for
   return archiveIsFixed

ArchiveFixContainmentCheck$ConstructFixContainmentCheck.containsFix():
   //Check the number of commits
   //Check the type of change ( MOD, ADD, DEL, NUL )
```

```
   //In case the change is MOD, check if the changes are the same across all paths
   OrderedCommitsForPath.getOverallChange()
      //check if the construct is fixed and type of the fix
      this.containsFix():
         /* there are four possible cases:
         1-added as part of the fix
           Considered fixed if the archive contains the construct
         2-deleted as part of the fix
           Considered fixed if the archive does not contain the construct
         3-modified as part of the fix
           Considered fixed if the archive contains the modified version of the construct
         4-didn't exist before and after
           Always consider as not fixed since the construct did not exist neither on the vulnerable
                or in the fixed version
         */
         if modified :
            //check if it is part of the archive and it contains the fix
            fixContained = OrderedCommitsForPath.containsFix()
         end if
         return fixContained

OrderedCommitsForPath.containsFix():
   //get the last and overall change to be compared to the signature under analysis
   OrderedCommitsForPath.getLastChange()
   //The overall change is obtained by comparing the vulnerable and fixed signature
   //Across all commits, only consider the changes introduced by the security commits
   OrderedCommitsForPath.getOverallChange()
   // Get and compare edit scripts Defective->Test and Test->Fixed
   // In order to compute Defective->Test, use the Decompiler
   // the decompiler creates a signature from the java class file
   //compare intersection between the edit scripts Defective->Test, and the intersection
        Test->Fixed
   i_dt = ASTUtil.intersectSourceCodeChanges(defective,test)
   i_tf = ASTUtil.intersectSourceCodeChanges(test,fixed)
   //The jar under test is vulnerable if i_dt == 0 AND i_tf > 0
   //The jar under test is fixed  if i_dt > 0 AND i_tf == 0
   if i_tf.isEmpty() AND i_dt.isNotEmpty():
      return true
   else:
      return false
   end if
```

A.2 Patch evaluator - Assess single library

```
assessLibrary(l):
    // look among the elements of the change list for bug b, finding classes, methods and
         constructors
    constructChanges = getConstructChanges(b)
    for cc in constructChanges:
        if cc is method or cc is constructor:
          if ( cc.changeType == MOD):
            // only if a method/constructor is modified as part of a security fix there are
                 fixedBody and vulnerableBody available
            affectedMethodsConstructors.add(cc)
      end if
    end if
  end for

    // for each method or constructor, try to perform the assessment
  for mc in affectedMethodsConstructors:
    if construnct cons in l:
      qnameInJar = true;
      if getSourcesForLibrary(l):
        // sources for library l are available
        sourcesAvailable = true;
        // retrieve the ast representation for the construct as found in current library
        ast_lid = getASTforConstruct(cons, l);
        if ast_lid is not null:
          buggyBody = getVulnerableVersionForConstruct(cons);
          fixedBody = getFixedVersionForConstruct(cons);
          // get the edit scripts for transforming the ast_lid into vulnerable and fixed version
               for construct cons
          buggyToLid = getEditScript(buggyBody, ast_lid);
          lidToFixed = getEditScript(ast_lid, fixedBody);
          if ( buggyToLid.isEmpty ):
            // code under test is exactly the same as it appears in vulnerable version
            construct_assessment = V;
            mc.setVulnerable();
          else if ( lidToFixed.isEmpty ):
            // code under test is exactly the same as it appears in fixed version
            construct_assessment = F;
            mc.setFixed();
          else:
            // code under test is different both from vulnerable and fixed version
            construct_assessment = N;
          end if
        else :
          // happens if it was not possible to retrieve the ast for the construct
          construct_assessment = X;
        end if
    else :
      // happens when library jar is not available for download
      construct_assessment = X;
    end if
  end for

  for mc in affectedMethodsConstructors:
    if ( mc.isVulnerable() ) :
      l.setVulnerable();
      break;
    else if ( mc.isFixed()):
      l.setFixed();
    end if
  end for
```

A.3 Patch Evaluator

```
AssessBug(b):
   // retrieve all the affected libraries from the backend
   libraries_ga = getAffectedLibrariesGA(b);
   // retrieve all the versions of all the affected libraries from Maven
   all_libraries = getAllLibraries(libraries_ga);
   // Phase 1
   // use basic approach to establish a first assessment
   for library l in all_libraries :
      assessSingleLibrary(l);
   end for

   // Phase 2
   // assessment using major release
   for library l in all_libraries :
      // get last vulnerable and first fixed version on the major release associated to library l
      lvff = getLastVulnerableFirstFixedPerMR(l.getMajorRelease());
      if ( lvff.exists() ) :
         if ( l <= lvff.getLastVulnerable() ) :
            l.setVulnerable();
         else if ( l >= lvff.getFirstFixed() ) :
            l.setFixed();
         end if
      end if
   end for

   // Phase 3
   // assessment using intersections
   for library l in all_libraries :
      // find the intersection for the major release associated to library l
      intersection = getIntersectionPerMR(l.getMajorRelease());
      // verify that intersection is unique on the major release
      verifyIntersection(intersection);
      if ( intersection.found() )
         if ( l <= intersection.getStart() )
            l.setVulnerable();
         else if ( l>= intersection.getEnd() )
            l.setFixed();
         end if
      end if
   end for

   // Attempt assessment for major release not affected
   // get the first release for each of the major releases
   firstReleasesPerMajorRelease = getFirstReleasePerMajorRelease();
   for library l in firstReleasesPerMajorRelease:
      // count how many libraries are assessed on the major release under analysis
      countAssessedLibsMajorRelease = countAssessedLibrariesPerMajorRelease();
      // check that no intersection exists for the major release
      intersection = getIntersectionPerMR(l.getMajorRelease());
      if ( intersection is null AND countFixedLibrariesPerMajorRelease == 0 ):
         // check that at least one intersection exists on the other major releases
         otherMRIntersections = getIntersectionsOnOtherMajorReleases(l.getMajorRelease());
         if ( otherMRIntersections.isNotEmpty() ):
            // get the latest intersection end available
            lastIntersectionEnd = getLastIntersectionEnd(otherMRIntersections);
            // if the first library of this major release is fixed, all the others will be fixed too
            if ( l > lastIntersectionEnd ):
               setFixedMajorRelease(l.getMajorRelease());
            end if
         end if
      end if
   end for
```

Appendix B

Metrics Evaluation Results

In this appendix I show the full tables obtained when running dependency finder and the simple metrics on archive *HttpComponents-HttpClient*, used for the analysis in Section 2.5.

B.1 Dependency Finder Results

Version	LVAR	OICF	OIPF	OIPC	OEPF	OEPC
4.0	7.0	0	0	0	12	6
4.2.3	6.0	0	0	0	15	6
4.2.5	6.0	0	0	0	15	6
4.3.5	3.0	1	0	0	2	4
4.5.2	4.0	0	1	0	2	4

Table B.1: Results of the evaluation at Method level, for method *getCns()*.

Version	M	A	IC	SUB	DOI	OIP	OEP
4.0	13.0	1.0	0.0	3.0	0.0	1	2
4.1.2	13.0	1.0	0.0	3.0	0.0	1	2
4.2.5	13.0	1.0	0.0	3.0	0.0	1	2
4.3	14.0	2.0	0.0	3.0	0.0	1	3
4.3.1	15.0	2.0	0.0	3.0	0.0	1	3
4.3.5	16.0	2.0	0.0	3.0	0.0	1	3
4.4	13.0	2.0	0.0	3.0	0.0	1	3
4.5.2	13.0	2.0	0.0	3.0	0.0	1	3

Table B.2: Results of the evaluation at Class level, for class *AbstractVerifier*.

Version	C	M	A	SUB	DOI
4.0	6.0	7.33	2.0	0.5	0.0
4.1	9.0	6.55	1.33	0.33	0.0
4.1.2	9.0	6.55	1.33	0.33	0.0
4.2	10.0	6.5	1.4	0.3	0.0
4.2.5	10.0	6.2	1.3	0.3	0.0
4.2.6	10.0	6.2	1.3	0.3	0.0
4.3	16.0	6.93	2.25	0.18	0.0
4.3.1	16.0	7.12	2.25	0.18	0.0
4.3.2	16.0	7.06	2.25	0.18	0.0
4.3.4	16.0	7.06	2.25	0.18	0.0
4.3.5	16.0	7.12	2.25	0.18	0.0
4.3.6	16.0	7.125	2.25	0.1875	0.0
4.4	18.0	7.72	2.55	0.16	0.0
4.5.2	20.0	7.25	2.55	0.15	0.0

Table B.3: Results of the evaluation at package level, for package *ssl*.

B.2 Simple metrics results

	4.0	4.0.1	4.0.2	4.0.3	4.1	4.1.1	4.1.2	4.1.3	4.2	4.2.1	4.2.2	4.2.3	4.2.4	4.2.5	4.2.6	4.3	4.3.1	4.3.2	4.3.3	4.3.4	4.3.5	4.3.6	4.4	4.4.1	4.5	4.5.1	4.5.2
4	1,00	1,00	1,00	1,00	0,89	0,89	0,88	0,81	0,81	0,81	0,81	0,81	0,81	0,81	0,67	0,67	0,67	0,67	0,67	0,67	0,67	0,62	0,62	0,62	0,62	0,62	0,62
4.0.1	1,00	1,00	1,00	1,00	0,89	0,88	0,88	0,81	0,81	0,81	0,81	0,81	0,81	0,81	0,67	0,67	0,67	0,67	0,67	0,67	0,67	0,62	0,62	0,62	0,62	0,62	0,62
4.0.2	1,00	1,00	1,00	1,00	0,89	0,88	0,88	0,82	0,81	0,81	0,81	0,81	0,81	0,81	0,67	0,67	0,67	0,67	0,67	0,67	0,67	0,62	0,62	0,62	0,62	0,62	0,62
4.0.3	1,00	1,00	1,00	1,00	0,89	0,88	0,88	0,82	0,81	0,81	0,81	0,81	0,81	0,81	0,67	0,67	0,67	0,67	0,67	0,67	0,67	0,62	0,62	0,62	0,62	0,62	0,62
4.1	0,89	0,89	0,89	0,89	1,00	1,00	0,99	0,91	0,90	0,90	0,90	0,90	0,90	0,90	0,74	0,74	0,74	0,74	0,74	0,74	0,74	0,68	0,68	0,68	0,68	0,68	0,68
4.1.1	0,89	0,88	0,89	0,89	1,00	1,00	0,99	0,91	0,90	0,90	0,90	0,90	0,90	0,90	0,74	0,74	0,74	0,74	0,74	0,74	0,74	0,68	0,68	0,68	0,68	0,68	0,68
4.1.2	0,88	0,88	0,88	0,88	0,99	1,00	1,00	0,92	0,92	0,92	0,91	0,90	0,90	0,90	0,74	0,74	0,74	0,74	0,74	0,74	0,74	0,68	0,68	0,68	0,68	0,68	0,68
4.1.3	0,88	0,88	0,88	0,88	0,99	1,00	1,00	0,92	0,92	0,92	0,91	0,90	0,90	0,90	0,74	0,74	0,74	0,74	0,74	0,74	0,74	0,68	0,68	0,68	0,68	0,68	0,68
4.2	0,81	0,81	0,82	0,82	0,91	0,92	0,92	1,00	1,00	1,00	0,99	0,99	0,99	0,98	0,81	0,81	0,81	0,81	0,81	0,81	0,81	0,75	0,75	0,75	0,75	0,75	0,75
4.2.1	0,81	0,81	0,81	0,81	0,90	0,91	0,91	1,00	1,00	1,00	1,00	0,99	0,99	0,98	0,81	0,81	0,81	0,81	0,81	0,81	0,81	0,75	0,75	0,75	0,75	0,75	0,75
4.2.2	0,81	0,81	0,81	0,81	0,90	0,91	0,91	1,00	1,00	1,00	1,00	0,99	0,99	0,98	0,81	0,81	0,81	0,81	0,81	0,81	0,81	0,75	0,75	0,75	0,75	0,75	0,75
4.2.3	0,81	0,81	0,81	0,81	0,90	0,90	0,90	0,98	0,99	0,99	0,99	1,00	1,00	0,99	0,82	0,81	0,81	0,81	0,81	0,81	0,81	0,75	0,75	0,75	0,75	0,75	0,75
4.2.4	0,81	0,81	0,81	0,81	0,90	0,90	0,90	0,98	0,99	0,99	0,99	1,00	1,00	0,99	0,82	0,81	0,81	0,81	0,81	0,81	0,81	0,75	0,75	0,75	0,75	0,75	0,75
4.2.5	0,81	0,81	0,81	0,81	0,90	0,90	0,90	0,98	0,98	0,98	0,98	0,99	0,99	1,00	0,82	0,81	0,81	0,81	0,81	0,81	0,81	0,75	0,75	0,75	0,75	0,75	0,75
4.2.6	0,81	0,81	0,81	0,81	0,90	0,90	0,89	0,97	0,98	0,98	0,98	0,99	0,99	1,00	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,75	0,75	0,75	0,75	0,75	0,75
4.3	0,67	0,67	0,67	0,67	0,74	0,74	0,74	0,81	0,81	0,81	0,82	0,82	0,82	0,82	1,00	1,00	1,00	1,00	0,98	0,98	0,98	0,91	0,91	0,91	0,91	0,91	0,91
4.3.1	0,67	0,67	0,67	0,67	0,74	0,74	0,74	0,81	0,81	0,81	0,81	0,81	0,81	0,81	1,00	1,00	1,00	1,00	0,98	0,98	0,98	0,91	0,91	0,91	0,91	0,91	0,91
4.3.2	0,67	0,67	0,67	0,67	0,74	0,74	0,74	0,80	0,81	0,81	0,81	0,81	0,81	0,81	1,00	1,00	1,00	1,00	0,99	0,99	0,99	0,91	0,91	0,91	0,91	0,91	0,91
4.3.3	0,67	0,67	0,67	0,67	0,74	0,74	0,74	0,80	0,81	0,81	0,81	0,81	0,81	0,81	1,00	1,00	1,00	1,00	0,99	1,00	1,00	0,93	0,93	0,93	0,93	0,92	0,92
4.3.4	0,67	0,67	0,67	0,67	0,74	0,74	0,74	0,80	0,81	0,81	0,81	0,81	0,81	0,81	0,98	0,98	0,98	0,98	0,99	1,00	1,00	0,93	0,93	0,93	0,93	0,92	0,92
4.3.5	0,67	0,67	0,67	0,67	0,74	0,74	0,74	0,81	0,81	0,81	0,81	0,81	0,81	0,82	0,98	0,98	0,99	0,99	0,99	1,00	1,00	0,93	0,93	0,93	0,93	0,92	0,92
4.3.6	0,67	0,67	0,67	0,67	0,74	0,74	0,74	0,81	0,81	0,81	0,81	0,81	0,81	0,82	0,98	0,98	0,99	0,99	0,99	1,00	1,00	0,93	0,93	0,93	0,93	0,92	0,92
4.4	0,62	0,62	0,62	0,62	0,68	0,68	0,68	0,74	0,75	0,75	0,75	0,75	0,75	0,75	0,91	0,91	0,91	0,92	0,93	0,93	0,93	1,00	1,00	0,99	0,99	0,99	0,98
4.4.1	0,62	0,62	0,62	0,62	0,68	0,68	0,68	0,74	0,75	0,75	0,75	0,75	0,75	0,75	0,91	0,91	0,91	0,92	0,93	0,93	0,93	1,00	1,00	0,99	0,99	0,99	0,98
4.5	0,62	0,62	0,62	0,62	0,68	0,68	0,68	0,74	0,75	0,75	0,75	0,75	0,75	0,75	0,91	0,91	0,91	0,92	0,93	0,93	0,93	1,00	1,00	1,00	1,00	0,99	0,99
4.5.1	0,62	0,62	0,62	0,62	0,68	0,68	0,68	0,74	0,75	0,75	0,75	0,75	0,75	0,75	0,91	0,91	0,91	0,91	0,92	0,92	0,92	0,99	0,99	1,00	1,00	1,00	1,00
4.5.2	0,62	0,62	0,62	0,62	0,68	0,68	0,68	0,74	0,75	0,75	0,75	0,75	0,75	0,75	0,91	0,91	0,91	0,91	0,92	0,92	0,92	0,99	0,99	1,00	1,00	1,00	1,00

Table B.4: Similarity for archive httpcomponents-httpclient.

	4.0	4.0.1	4.0.2	4.0.3	4.1	4.1.1	4.1.2	4.1.3	4.2	4.2.1	4.2.2	4.2.3	4.2.4	4.2.5	4.2.6	4.3	4.3.1	4.3.2	4.3.3	4.3.4	4.3.5	4.3.6	4.4	4.4.1	4.5	4.5.1	4.5.2
4	1,00	1,00	1,00	1,00	0,89	0,89	0,88	0,88	0,81	0,81	0,81	0,81	0,81	0,81	0,81	0,67	0,67	0,67	0,67	0,67	0,67	0,67	0,62	0,62	0,62	0,62	0,62
4.0	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	0,96	0,92	0,92	0,92	0,92	0,88	0,88	0,82	0,82	0,82	0,82	0,82
4.0.1	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	0,96	0,92	0,92	0,92	0,92	0,88	0,88	0,82	0,82	0,82	0,82	0,82
4.0.2	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	0,96	0,92	0,92	0,92	0,92	0,88	0,88	0,82	0,82	0,82	0,82	0,82
4.0.3	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	0,96	0,92	0,92	0,92	0,92	0,88	0,88	0,82	0,82	0,82	0,82	0,82
4.1	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	0,96	0,92	0,92	0,92	0,92	0,88	0,88	0,82	0,82	0,82	0,82	0,82
4.1.1	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	0,96	0,92	0,92	0,92	0,92	0,88	0,88	0,82	0,82	0,82	0,82	0,82
4.1.2	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	0,96	0,92	0,92	0,92	0,92	0,88	0,88	0,82	0,82	0,82	0,82	0,82
4.1.3	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	0,96	0,92	0,92	0,92	0,92	0,88	0,88	0,82	0,82	0,82	0,82	0,82
4.2	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	0,96	0,92	0,92	0,92	0,92	0,88	0,88	0,82	0,82	0,82	0,82	0,82
4.2.1	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	0,96	0,92	0,92	0,92	0,92	0,88	0,88	0,82	0,82	0,82	0,82	0,82
4.2.2	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	0,96	0,92	0,92	0,92	0,92	0,88	0,88	0,82	0,82	0,82	0,82	0,82
4.2.3	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	0,96	0,92	0,92	0,92	0,92	0,88	0,88	0,82	0,82	0,82	0,82	0,82
4.2.4	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	0,96	0,92	0,92	0,92	0,92	0,88	0,88	0,82	0,82	0,82	0,82	0,82
4.2.5	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	0,96	0,92	0,92	0,92	0,92	0,88	0,88	0,82	0,82	0,82	0,82	0,82
4.2.6	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	1,00	0,96	0,92	0,92	0,92	0,92	0,88	0,88	0,82	0,82	0,82	0,82	0,82
4.3	0,96	0,96	0,96	0,96	0,96	0,96	0,96	0,96	0,96	0,96	0,96	0,96	0,96	0,96	0,96	1,00	0,96	0,96	0,96	0,96	0,92	0,92	0,78	0,78	0,78	0,78	0,78
4.3.1	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,96	1,00	1,00	1,00	1,00	0,96	0,96	0,75	0,75	0,75	0,75	0,75
4.3.2	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,96	1,00	1,00	1,00	1,00	0,96	0,96	0,75	0,75	0,75	0,75	0,75
4.3.3	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,96	1,00	1,00	1,00	1,00	0,96	0,96	0,75	0,75	0,75	0,75	0,75
4.3.4	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,92	0,96	1,00	1,00	1,00	1,00	0,96	0,96	0,75	0,75	0,75	0,75	0,75
4.3.5	0,88	0,88	0,88	0,88	0,88	0,88	0,88	0,88	0,88	0,88	0,88	0,88	0,88	0,88	0,88	0,92	0,96	0,96	0,96	0,96	1,00	1,00	0,72	0,72	0,72	0,72	0,72
4.3.6	0,88	0,88	0,88	0,88	0,88	0,88	0,88	0,88	0,88	0,88	0,88	0,88	0,88	0,88	0,88	0,92	0,96	0,96	0,96	0,96	1,00	1,00	0,72	0,72	0,72	0,72	0,72
4.4	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,78	0,75	0,75	0,75	0,75	0,72	0,72	1,00	1,00	1,00	1,00	1,00
4.4.1	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,78	0,75	0,75	0,75	0,75	0,72	0,72	1,00	1,00	1,00	1,00	1,00
4.5	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,78	0,75	0,75	0,75	0,75	0,72	0,72	1,00	1,00	1,00	1,00	1,00
4.5.1	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,78	0,75	0,75	0,75	0,75	0,72	0,72	1,00	1,00	1,00	1,00	1,00
4.5.2	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,82	0,78	0,75	0,75	0,75	0,75	0,72	0,72	1,00	1,00	1,00	1,00	1,00

Table B.5: Similarity for class AbstractVerifier.

	4.0	4.0.1	4.0.2	4.0.3	4.1	4.1.1	4.1.2	4.1.3	4.2	4.2.1	4.2.2	4.2.3	4.2.4	4.2.5	4.2.6	4.3	4.3.1	4.3.2	4.3.3	4.3.4	4.3.5	4.3.6	4.4	4.4.1	4.5	4.5.1	4.5.2
4.0	1.00	1.00	1.00	1.00	0.78	0.78	0.78	0.70	0.70	0.70	0.70	0.71	0.71	0.71	0.71	0.50	0.48	0.48	0.48	0.48	0.47	0.47	0.43	0.43	0.43	0.43	0.41
4.0.1	1.00	1.00	1.00	1.00	0.78	0.78	0.78	0.70	0.70	0.70	0.70	0.71	0.71	0.71	0.71	0.50	0.48	0.48	0.48	0.48	0.47	0.47	0.43	0.43	0.43	0.43	0.41
4.0.2	1.00	1.00	1.00	1.00	0.78	0.78	0.78	0.70	0.70	0.70	0.70	0.71	0.71	0.71	0.71	0.50	0.48	0.48	0.48	0.48	0.47	0.47	0.43	0.43	0.43	0.43	0.41
4.0.3	1.00	1.00	1.00	1.00	0.78	0.78	0.78	0.70	0.70	0.70	0.70	0.71	0.71	0.71	0.71	0.50	0.48	0.48	0.48	0.48	0.47	0.47	0.43	0.43	0.43	0.43	0.41
4.1	0.78	0.78	0.78	0.78	1.00	1.00	1.00	0.92	0.92	0.92	0.92	0.92	0.92	0.92	0.92	0.57	0.55	0.55	0.55	0.55	0.55	0.55	0.51	0.51	0.51	0.51	0.48
4.1.1	0.78	0.78	0.78	0.78	1.00	1.00	1.00	0.92	0.92	0.92	0.92	0.92	0.92	0.92	0.92	0.57	0.55	0.55	0.55	0.55	0.55	0.55	0.51	0.51	0.51	0.51	0.48
4.1.2	0.78	0.78	0.78	0.78	1.00	1.00	1.00	0.92	0.92	0.92	0.92	0.92	0.92	0.92	0.92	0.57	0.55	0.55	0.55	0.55	0.55	0.55	0.51	0.51	0.51	0.51	0.48
4.1.3	0.78	0.78	0.78	0.78	1.00	1.00	1.00	0.92	0.92	0.92	0.92	0.92	0.92	0.92	0.92	0.57	0.55	0.55	0.55	0.55	0.55	0.55	0.51	0.51	0.51	0.51	0.48
4.2	0.70	0.70	0.70	0.70	0.92	0.92	0.92	1.00	1.00	1.00	1.00	0.99	0.99	0.99	0.99	0.64	0.63	0.63	0.63	0.63	0.62	0.62	0.58	0.58	0.58	0.58	0.55
4.2.1	0.70	0.70	0.70	0.70	0.92	0.92	0.92	1.00	1.00	1.00	1.00	0.99	0.99	0.99	0.99	0.64	0.63	0.63	0.63	0.63	0.62	0.62	0.58	0.58	0.58	0.58	0.55
4.2.2	0.70	0.70	0.70	0.70	0.92	0.92	0.92	1.00	1.00	1.00	1.00	0.99	0.99	0.99	0.99	0.64	0.63	0.63	0.63	0.63	0.62	0.62	0.58	0.58	0.58	0.58	0.55
4.2.3	0.70	0.70	0.70	0.70	0.92	0.92	0.92	1.00	1.00	1.00	1.00	0.99	0.99	0.99	0.99	0.64	0.63	0.63	0.63	0.63	0.62	0.62	0.58	0.58	0.58	0.58	0.55
4.2.4	0.71	0.71	0.71	0.71	0.92	0.92	0.92	0.99	0.99	0.99	0.99	1.00	1.00	1.00	1.00	0.65	0.63	0.63	0.63	0.63	0.63	0.63	0.58	0.58	0.58	0.58	0.56
4.2.5	0.71	0.71	0.71	0.71	0.92	0.92	0.92	0.99	0.99	0.99	0.99	1.00	1.00	1.00	1.00	0.65	0.63	0.63	0.63	0.63	0.63	0.63	0.58	0.58	0.58	0.58	0.56
4.2.6	0.71	0.71	0.71	0.71	0.92	0.92	0.92	0.99	0.99	0.99	0.99	1.00	1.00	1.00	1.00	0.65	0.63	0.63	0.63	0.63	0.63	0.63	0.58	0.58	0.58	0.58	0.56
4.3	0.50	0.50	0.50	0.50	0.57	0.57	0.57	0.64	0.64	0.64	0.64	0.65	0.65	0.65	1.00	0.98	0.98	0.98	0.98	0.98	0.98	0.98	0.90	0.90	0.90	0.90	0.87
4.3.1	0.48	0.48	0.48	0.48	0.55	0.55	0.55	0.63	0.63	0.63	0.63	0.63	0.63	0.63	0.98	1.00	1.00	1.00	1.00	1.00	0.99	0.99	0.89	0.89	0.89	0.89	0.85
4.3.2	0.48	0.48	0.48	0.48	0.55	0.55	0.55	0.63	0.63	0.63	0.63	0.63	0.63	0.63	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.89	0.89	0.89	0.89	0.86
4.3.3	0.48	0.48	0.48	0.48	0.55	0.55	0.55	0.63	0.63	0.63	0.63	0.63	0.63	0.63	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.89	0.89	0.89	0.89	0.86
4.3.4	0.48	0.48	0.48	0.48	0.55	0.55	0.55	0.63	0.63	0.63	0.63	0.63	0.63	0.63	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.89	0.89	0.89	0.89	0.86
4.3.5	0.47	0.47	0.47	0.47	0.55	0.55	0.55	0.62	0.62	0.62	0.62	0.63	0.63	0.63	0.98	0.99	1.00	1.00	1.00	1.00	1.00	1.00	0.88	0.88	0.88	0.88	0.85
4.3.6	0.47	0.47	0.47	0.47	0.55	0.55	0.55	0.62	0.62	0.62	0.62	0.63	0.63	0.63	0.98	1.00	1.00	1.00	1.00	1.00	1.00	1.00	0.88	0.88	0.88	0.88	0.85
4.4	0.43	0.43	0.43	0.43	0.51	0.51	0.51	0.58	0.58	0.58	0.58	0.58	0.58	0.58	0.90	0.89	0.89	0.89	0.89	0.89	0.88	0.88	1.00	1.00	1.00	1.00	0.97
4.4.1	0.43	0.43	0.43	0.43	0.51	0.51	0.51	0.58	0.58	0.58	0.58	0.58	0.58	0.58	0.90	0.89	0.89	0.89	0.89	0.89	0.88	0.88	1.00	1.00	1.00	1.00	0.97
4.5	0.43	0.43	0.43	0.43	0.51	0.51	0.51	0.58	0.58	0.58	0.58	0.58	0.58	0.58	0.90	0.89	0.89	0.89	0.89	0.89	0.88	0.88	1.00	1.00	1.00	1.00	0.97
4.5.1	0.43	0.43	0.43	0.43	0.51	0.51	0.51	0.58	0.58	0.58	0.58	0.58	0.58	0.58	0.90	0.89	0.89	0.89	0.89	0.89	0.88	0.88	1.00	1.00	1.00	1.00	0.97
4.5.2	0.41	0.41	0.41	0.41	0.48	0.48	0.48	0.55	0.55	0.55	0.55	0.56	0.56	0.56	0.87	0.85	0.86	0.86	0.86	0.86	0.85	0.85	0.97	0.97	0.97	0.97	1.00

Table B.6: Similarity for package ssl.

Appendix C

Frontend

The frontend is a web application used to show in an easily understandable way, for each bug known by vulas, the assessments of the potentially affected libraries.

As a first page, the list of all the bugs known by vulas is shown, from which one bug can be selected and then obtain the results of the assessments.

The list of all the libraries is a combination of all the versions known to maven and all the archives been analyzed by the version check and patch evaluator approaches. For each vulnerability the user can see the result of the assessments performed by version check, patch evaluator or manually and, if he knows the assessment for a certain version, manually set it. This is very useful when no tool has produced any assessment for the library, or if the produced assessment is wrong. Part of the list and assessment for the libraries of a certain bug are shown in Figure C.1.

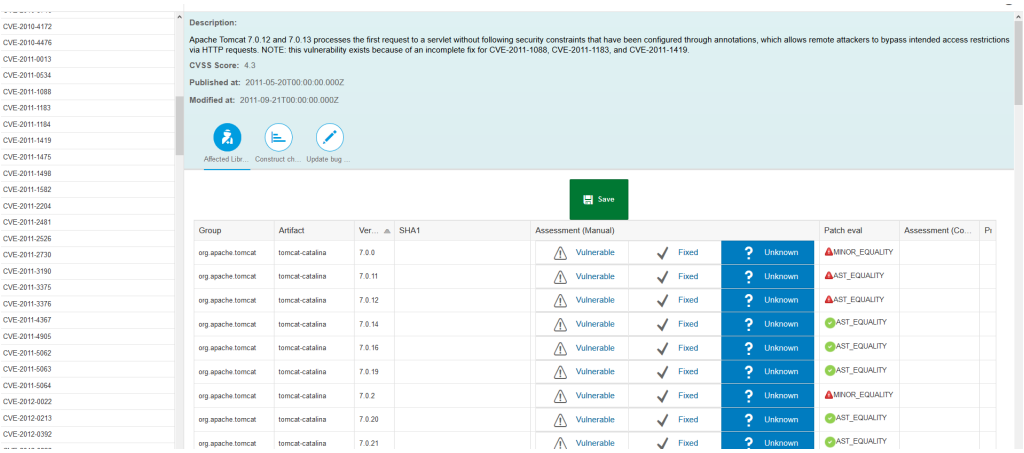


Figure C.1: Bugs list and assessment for libraries of a certain bug.

Change list page

A dedicated page is used in order to better understand which were the changes performed in order to fix a the bug under analysis, and all the construct changes are shown in two table: in the first one divided by commit identifier and, in the second one, divided by path. An example of this page is shown in Figure C.2.

If a construct has been modified as part of the fix, it is interesting to look at the changes performed, therefore a page with three tables has been created:

1. Overall change table: this table contains all the changes performed on the selected construct in order to fix the vulnerability.

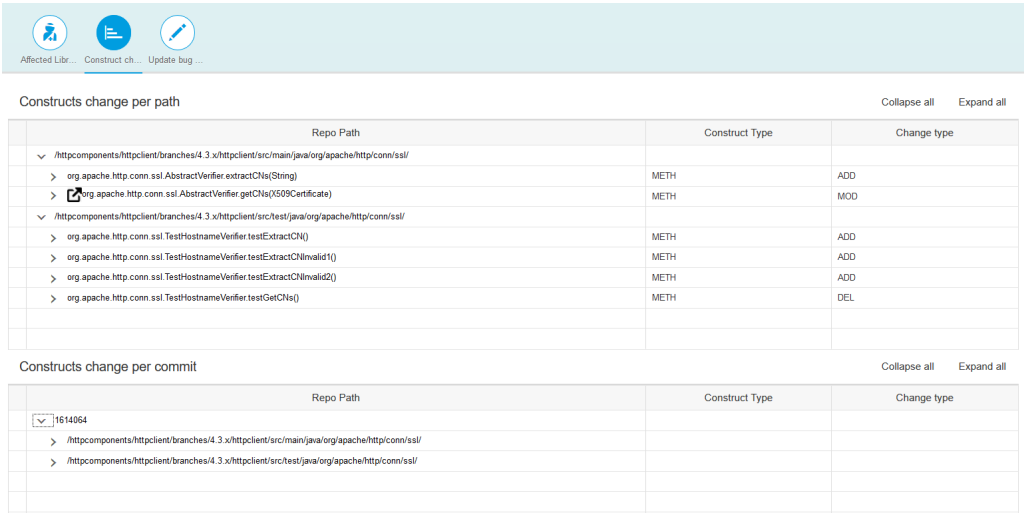


Figure C.2: Change list of a certain bug.

2. Vulnerable version table: shows, as a tree, the vulnerable version of the selected construct;
3. Fixed version table: shows, as a tree, the fixed version of the selected construct;

For an easier understanding, all the rows that have been modified are attached with an identifier (either MOD, ADD or DEL) in all the tables described above. It is also possible for the user to click on every row of every table, highlighting the correspondent row in the other tables. An example of this page is shown in Figure C.3.

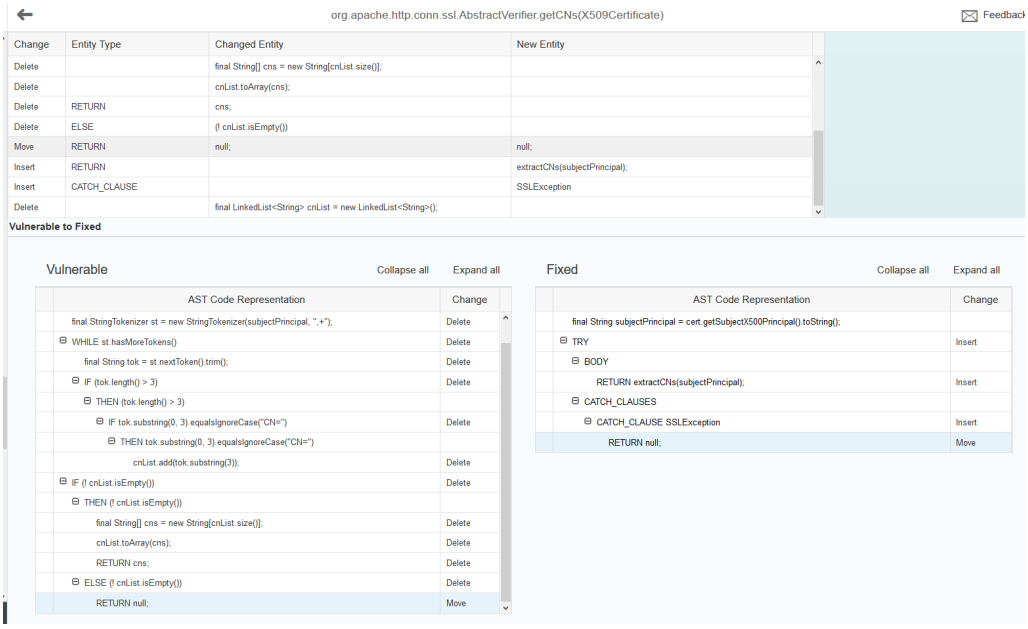


Figure C.3: Change list, vulnerable and fixed version of a construct part of an archive affected by a certain bug.

Patch evaluator details page

In case a library has been assessed by patch evaluator, it is possible for the user to click on it and get more details in order to know how the decision has been taken. For each of the modified constructs from the change list of the bug under analysis, it is possible to see whether it is equal

or not to its vulnerable or fixed version, and the distance to both versions. According to the phase in which the assessment for the selected archive was done, it is possible to see information about:

- Last vulnerable and first fixed version (if any) for the major release that the archive belongs to;
- The intersection (if any) found on the major release that the archive belongs to.

An example of this page is shown in Figure C.4.

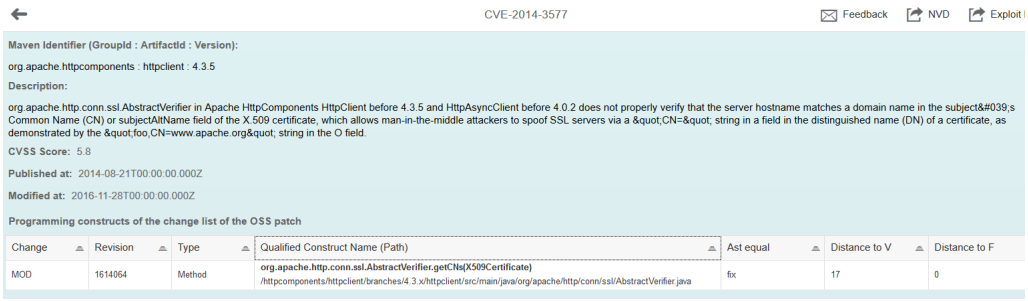


Figure C.4: Bugs list and assessment for libraries of a certain bug.

Version check details page

In case a library has been assessed by version check, it is possible for the user to click on it and get more details in order to know how the decision has been taken. For each of the modified constructs from the change list of the bug under analysis, it is possible to see whether it has been found fixed or vulnerable, if it is contained or not in the selected archive, if the class it belongs to is contained or not in the selected archive. An example of this page is shown in Figure C.5.

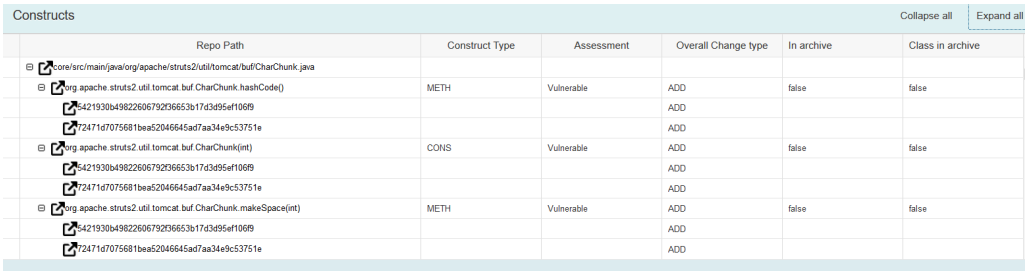


Figure C.5: Bugs list and assessment for libraries of a certain bug.

Bibliography

[1] Sonatype survey, http://blog.sonatype.com/2010/12/now-available-central-download-statistics-for-oss-projects/
[2] Heartbleed bug, http://heartbleed.com/
[3] The OpenSSL project, http://www.openssl.org/
[4] The Open Web Application Security Project website, https://www.owasp.org
[5] Owasp Top-10 for 2013, https://www.owasp.org/index.php/Top_10_2013-A9-Using_Components_with_Known_Vulnerabilities
[6] Owasp Dependency check, https://www.owasp.org/index.php/OWASP_Dependency_Check
[7] Plate, Henrik, Serena Elisa Ponta, and Antonino Sabetta, "Impact assessment for vulnerabilities in open-source software libraries", Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on. IEEE, 2015.
[8] Procyon decompiler, https://bitbucket.org/mstrobel/procyon/wiki/Home
[9] Wala, http://wala.sourceforge.net/wiki/index.php/Main_Page
[10] Soot, https://sable.github.io/soot/
[11] Abstract Syntax Tree, Wikipedia definition, https://en.wikipedia.org/wiki/Abstract_syntax_tree
[12] ChangeDistiller website, https://bitbucket.org/sealuzh/tools-changedistiller/wiki/Home
[13] FLURI, Beat, et al. Change distilling: Tree differencing for fine-grained source code change extraction. IEEE Transactions on Software Engineering, 2007, 33.11: 725-743.
[14] Apache POI library, http://poi.apache.org/changes.html
[15] Levenshtein distance, https://en.wikipedia.org/wiki/Levenshtein_distance
[16] Apache Maven, software project management and comprehension tool, https://maven.apache.org/
[17] NVD details for CVE-2014-0095, https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0095
[18] Maven Central website, http://search.maven.org/
[19] Control Flow Graph, https://en.wikipedia.org/wiki/Control_flow_graph
[20] FERRANTE, Jeanne; OTTENSTEIN, Karl J.; WARREN, Joe D. The program dependence graph and its use in optimization. ACM Transactions on Programming Languages and Systems (TOPLAS), 1987, 9.3: 319-349.
[21] SHA-1 algorithm, https://en.wikipedia.org/wiki/SHA-1