Gottfried Wilhelm
Leibniz Universität Hannover
Faculty of Electrical Engineering and Computer Science
Institute of Practical Computer Science
Software Engineering Group

Security Study with the Use of Known Vulnerabilities in Github

Master Thesis

in Computer Science

by

Simon van Schwartzenberg

First Examiner: Prof. Dr. Kurt Schneider
Second Examiner: Dr. Jil Klünder
Supervisor: M. Sc. Fabien Patrick Viertel

Hannover, February 17, 2020

Abstract

When writing software, developers tend to be unaware, when they have produced dangerous code, which can lead to severe consequences. This is, why increasing effort is put into the development of the detection of such code. Developers are rarely security experts, and changing this would be unrealistic. Because of this, most work is put into automated approaches, where the expertise of the developers is of no concern. These approaches mostly use community knowledge to identify vulnerable code and provide fixes. This thesis provides a study analyzing, to what extent software publicly developed on GitHub.com is still being developed with vulnerabilities. This is done by analyzing entire development histories of several projects hosted on GitHub.com, written in the programming languages Python, Java and JavaScript. The tool for this analysis is based upon a plugin for JIRA, that was used to apply security code clone detection to freshly committed changes regarding source code written in Java and JavaScript. For this thesis, the software was disconnected from JIRA, changed to apply to all commits of a project, and extended with a module for Python-related detection - to name the most significant changes. Extensive data has been gathered and analyzed about every single commit done to a total of 159 projects on GitHub.com and five main research questions about vulnerability statistics of the three languages have been answered, along with a few more deductions taken from the result data.

Contents

1 Introduction

1.1 Motivation

In the early phases of software development, a decision for a code language has to be made. In this thesis, the three languages Python, Java, and JavaScript are targeted and compared in the context of vulnerabilities, to add differing susceptibility to vulnerabilities to the list of factors for that decision. What can emerge, is that one language is more vulnerable to, for example, Cross-Site Scripting (XSS), than the other two. This information can shift the decision for a language in case the software is needed to be proof against XSS in particular. To learn more about vulnerabilities and improve tactics to avoid them, they need to be found first. This thesis provides a study of vulnerabilities regarding the most prominent public projects on GitHub.com, which is created using a tool that has been written with the intent of also supporting further studies.

1.2 Goals

The intention of this thesis and its study is to answer five research questions. This will be achieved by building a tool, that combines Library Checker[21] (LC) and Security Code Clone Detector[19] (CCD) into one, adding keyword detection to the mix, and generating a dataset that contains lots of data about possible vulnerabilities in the most prominent repositories on GitHub.com.

RQ1: How many applications contain security code clones?
To answer this, a sufficient number of applications are scanned for vulnerabilities, and the portion of them containing vulnerabilities is determined.

RQ2: Which programming languages are susceptible to which Common Weakness Enumeration[16] (CWE)?
This question spreads out to 6 partly questions, as each of the 3 targeted languages will have results for both LC and CCD, and will be answered thrice. Both modules, LC and CCD will be separately discussed with generated data about the most frequent vulnerability classes of each language, identified by so-called CWE-ids. After that is done, in the third answer, the results will be compared, to find, if the modules agree one some emergences.

RQ3: Which coding language has more vulnerabilities?
For this, statistical values are created using our tool, to compare, how often code of our

three languages contains vulnerabilities among the scanned repositories.

RQ4: Which libraries introduce vulnerabilities into software most commonly?
This one is only answerable using the results of the LC module, as it is the only one working with libraries. To answer this question, the libraries that emerged as vulnerable across all repositories are counted and the three most frequent occurrences are discussed for each language.

RQ5: Is the size of a repository related to its susceptibility to vulnerabilities?
For this question, there are, again, answers from both, the CCD and the LC. The number of vulnerability occurrences in each repository is inserted into a graph, together with a measure of the size of each repository. Then, potential trends are discussed for the vulnerability frequency in the repositories.

Additionally, the result set is analyzed to answer more questions that come to mind. For example, the results of the keyword detection are not represented in the five main questions but might still show interesting results when connected to the other two main modules.

1.3 Structure of this Thesis

This thesis consists of eight chapters, including this introduction. In the second chapter "Basics" the integral terms of the topic are explained. This includes the mechanics of the tool and related software. Then, in the third chapter, the structure, modules, and processes of the tool are elucidated. In the fourth chapter, the implementation of the software is described in more technical detail and the libraries, the framework, and technical quirks used are explained. The fifth chapter, "Results and Discussion", is the chapter handling the validity of the tool, its results, their discussion, answers to the research questions, and threats to validity. The related work chapter will then mention various papers and theses regarding previous work, similar approaches, and resources for future improvement. In the seventh chapter, "Conclusion", the success of this thesis and the accompanying study are discussed. Lastly, in the eighth chapter, "Future Work" various suggestions are given, to improve the developed tool and its concept for future studies.

2 Basics

In the following chapter the basics of this thesis will be elucidated, such as the data sources worked with, the techniques utilized and the terminology used in the context of this thesis.

2.1 Vulnerabilities

Vulnerabilities are the main target of this thesis.
ISO 27005[4] defines vulnerability as:
"A weakness of an asset or group of assets that can be exploited by one or more threats."
IETF RFC 4949[5] defines vulnerability as:
"A flaw or weakness in a system's design, implementation, or operation and management that could be exploited to violate the system's security policy."
For the context of Common Vulnerabilities and Exposures[15] (CVE), Vulnerabilities are defined like this by National Institute of Standards and Technology[26] (NIST):
"A weakness in the computational logic (e.g., code) found in software and hardware components that, when exploited, results in a negative impact to confidentiality, integrity, or availability. Mitigation of the vulnerabilities in this context typically involves coding changes, but could also include specification changes or even specification deprecations (e.g., removal of affected protocols or functionality in their entirety)."[44]

2.2 National Vulnerability Database

The National Vulnerability Database (NVD) is a public database created in the year 2000[37] by the NIST Computer Security Division[38], Information Technology Laboratory[39] and is sponsored by the U.S. Department of Homeland Security's National Cyber Security Division[40]. It is built upon the CVE-list that was created in 1999 and since tended to by "Mitre Corporation"[41]. It is a reference for numerous reports of software vulnerabilities. Mitre and NIST both provide public datasets with synchronized data, but there are differences. NIST's NVD offers a public downloadable feed of the whole database in JSON format. It used to be available in XML as well, but that feed has been discontinued in the "XML Vulnerability Retirement Update" of October 9th, 2019[42]. Furthermore, the NVD is a superset of the Mitre list in such a way, that in the NVD

the data is enriched with further analysis including the combination of the following four identification/classification standards.

2.2.1 CVE

Common Vulnerabilities and Exposures[15] (CVE) is a standard to identify reported vulnerabilities and exposures in computer systems. For example, the following is the identification URI of the first CVE ever reported:
"CVE-1999-0001"
This key identifies - besides the fact that it is the first vulnerability report ever - the first vulnerability that was reported in the year 1999 and is further defined in the NVD. There you can read a description of the CVE entry, its relations to Common Platform Enumeration[14] (CPE), Common Weakness Enumeration[16] (CWE), its Common Vulnerability Scoring System[17] (CVSS) rating, references to the vendor's homepage and public reports of the vulnerability.

2.2.2 CWE

One Common Weakness Enumeration[16] (CWE) entry can be identified by a simple integer id. The difference to CVE is that the CWE-context is more abstract and acts more like a category of weaknesses. If, for instance, you had a total of 200 CVE reports for 200 distinct CPE entries regarding SQL injection they might all reference CWE-89. Not all CVE reports have an assigned CWE while some have multiple. This can mean that there is no CWE fitting the report, but since the assignment has to be done manually, the great majority likely just hasn't been assigned yet. The Mitre CWE database currently (as of 12.01.2020) contains 808 weaknesses and many CWE entries that are not weaknesses themselves but are used to combine and categorize the weaknesses. There is, for example, the previously mentioned
"CWE-89: Improper Neutralization of Special Elements used in an SQL Command ('SQL Injection')"[72]
It is a proper weakness category - not a category of CWE entries - and is referenced by - according to an NVD dump created throughout this study - 7.609 CVE reports. CWE-89 itself is a member of 15 higher-level CWE categorizations.

2.2.3 CPE

Common Platform Enumeration[14] (CPE) is a standardized format to identify - not exclusively - software. It is used by the NVD in combination with CVE, CVSS, and CWE. The following is an example of a CPE key:
"cpe:2.3:o:microsoft:windows_xp:-:-:*:*:professional:*:x64:*"
It is using the following format:

"cpe:cpeversion:part:vendor:product:version:update:edition:...
...language:sw_edition:target_sw:target_hw:other"
Where "part" defines if the CPE key describes an application, an operating system or hardware, "vendor", "product" and "version" are the phrases that are usually filled, sufficiently defining the product. The other fields are very optional and barely used at all.

2.2.4 CVSS

Common Vulnerability Scoring System[17] (CVSS) is a standard used to rate CVE reports. This is the one standard used in the NVD that does not have its own identification because it is contained within each CVE. One set contains a total of fifteen fields, enabling a precise rating of the vulnerability. This includes the way a vulnerability can be exploited, its severity, obtained privileges on a system, if a user has to be involved, and more. Currently, CVSS version 2[17] (CVSSV2) and CVSS version 3[18] (CVSSV3) are the used standards. CVSSV3 is not always present though, whereas CVSSV2 is, leading to single CVE sometimes having both. These two ratings for a single entry tend to have differing values for the fields they share, so we will, for our study, settle for only using CVSSV2. Here is an example for a representative vector string for a CVSSV2 rating:
"AV:L/AC:L/Au:N/C:C/I:C/A:C"
This is a shortened form of a part of the rating — not the whole rating. It translates to:

- "accessVector" : "LOCAL",
- "accessComplexity" : "LOW",
- "authentication" : "NONE",
- "confidentialityImpact" : "COMPLETE",
- "integrityImpact" : "COMPLETE",
- "availabilityImpact" : "COMPLETE",

These six fields are combined into three numeric ratings, called exploitabilityScore, impactScore and baseScore. The exploitabilityScore of the excerpts is a combination of the fields "accessVector", "accessComplexity", and "authentication". These three fields are assigned with a selection of values, for example, "accessVector" can be "requires local access", "adjacent network accessible", or "network accessible". Each of those values is assigned with a fixed value between 0 and 1. The score is then calculated as follows:

$$exploitabilityScore = accessVector \cdot accessComplexity \cdot authentication \cdot 20$$

The values are chosen in such a way, that this equation will not exceed a score of

10 - this holds true for all three scores. The score is meant to rate how difficult the vulnerability is to abuse, with a high value indicating easy prey.
The impactScore is based on the fields confidentialityImpact, integrityImpact, and availabilityImpact and is calculated like this:

$$impactScore = 10,41 \cdot (1 - (1 - confidentialityImpact) \cdot (1 - integrityImpact) \cdot (1 - availabilityImpact))$$

It is representing the impact on the three key concepts of information security, with a high score implying bad consequences for the system.
The baseScore is a combination of the other two, with the impactScore being weighed higher than the exploitabilityScore. The formula for it is the following (rounded to one decimal):

$$baseScore = ((0,6 \cdot impactScore) + (0,4 \cdot exploitabilityScore) - 1,5) \cdot f(impactScore)$$

$f(impactScore)$ is 0 if impactScore is 0, and else it is 1,176, making the impactScore being larger than 0 mandatory for any baseScore other than 0, while an exploitabilityScore of 0 is no issue.

2.3 Library Checker

The Library Checker[21] (LC) is one of the heart pieces of this thesis and provides half of the data, albeit being much simpler than the other heart piece - the Security Code Clone Detector[19] (CCD). The LC's ambition is extracting the names and versions of the libraries used in the projects in question and use those to search the NVD for reports of related vulnerabilities and exploits. This is done in the following two steps:
First, it needs a way to extract the names of the libraries used with as much recall as possible. In this part, precision is usually of no concern, because a string is either a library identification or not. If the LC extracts a string as library identification that is not meant as such, the LC is quite frankly just broken. Even if it does, the name would still have to match a CPE in the NVD to be treated as a match. These identifications can be extracted from a centralized dependency management system, like Node Package Manager[20] (NPM) for JavaScript or Maven[35] for Java. Using these, one has a chance to get a more detailed identification, possibly including the precise version of the library used and a reference to where it is updated from. What's so simple about the LC, is that the name of a library either matches a CPE in its entirety, or it does not. There is no partial string similarity to calculate and consider. Should the CPE have been reported with a name that one will not find through a query, then that is where the mistake has been made.

2.4 GitHub

GitHub[33] is a company and public platform providing hosting for software development version control through Git[34]. Git is essentially used to track changes to a project, also by multiple simultaneous workers if needed. In this thesis, GitHub.com is of twofold importance. First, it is used in one of the extraction tools covered later in this chapter. Second, it is used as the focus of this study, as all projects scanned for the study are managed by and drawn from GitHub.com.

2.4.1 Structure

GitHub.com[33] contains many kinds of datasets. When working with GitHub.com, in most use-cases, you first need to create a user profile. It contains references to one's own created repositories, commits made, issues and comments authored, starred repositories, some statistics, etc.. The next step is creating a "repository", which is a pivotal reference for the entirety of the data GitHub will have about a project. It contains the files created and committed to it, as well as reported issues, statistics, permissions for user profiles, as well as a log of every single committed change made to the project. Such a change is called a commit. A commit consists of the changes made in comparison to the previous commit and a commit message. A commit can, but usually will not, contain changes to a single file only. Rather than having each commit contain the entire project at that respective state, each commit builds upon its predecessor commit. A user can browse public repositories and "star" repositories he deems interesting. "Starring" a repository works like a bookmark, linking an account to a repository. The number of users that have bookmarked a repository in this way is displayed on the repository's main page as "stars"[43].

2.4.2 API

GitHub provides an online URL-Based Application Programming Interface[28] (API) to work with.[11] With it, one can query from for example a list of all repositories, filtered as requested. The following is an example of the URLs used in the software of this thesis.

```
https://api.github.com/search/repositories?q=language:Python&
private:false&sort=stars&order=desc&page=1&per_page=1
```

This resembles a query to a database — because it is used as such. This URL asks GitHub.com for their repository with the most stars for the language tagged as using Python code that is not private. One could request a broad range of data from GitHub.com this way, such as public user data or - for this thesis more important - rate-

limit-related information regarding the requester's IP-address. The rate-limit information is utilized in the tool as GitHub limits the number of requests a user is permitted to make per fixed timespan. GitHub is currently - as of late 2019 - working to replace this API with a query language called GraphQL[12] in the so-called GraphQL API v4.[13] This new API has not been used here because it is currently very sparsely documented, but in the future, the old API might be deactivated entirely.

2.5 ANTLR

ANother Tool for Language Recognition[10] (ANTLR)[10] is a parser generator. To clarify first, two groups of languages are distinguished here: One group consists of the code generation target languages.[3] They are the languages for which the resulting code will be executable. The other group consists of the parser target languages.[9] These languages are the repertoire that our parser, which was generated by ANTLR, is capable of parsing into Abstract Syntax Trees[23] (ASTs). Currently, there are nine code generation languages, including Java - which is the one used in our tool. There is a repository[9] on GitHub.com showing template files for an abundance of parser target languages. In the case of the software of this study the template used was for Python3.[2] The generated code can - depending on the integrity/quality of the template file - build an AST from any source code of the respective language, that a fitting compiler would accept as correct code. Having an AST as a structure for the code clone detection opens up an opportunity to build a very precise and configurable algorithm, which is not currently done in this thesis, but discussed in the future work chapter. Instead, the AST is stripped of its structure and dissected into tokens.

2.6 Tokenizer

In the case of Python, code is parsed into an AST using ANTLR. An example of such an AST can be seen in figure 2.1. This AST contains several nodes one can pick to isolate subtrees, and thus, code blocks of the desired granularity. If for example, the desired granularity were to be at "method" level, the nodes of the respective classification can be located and used as root nodes. From those root nodes on, all other nodes can be ignored down until the leaves of the subtree. These leaves are the words written in the code and the result would be one token set per isolated "root" node.

Noise Characters

Tokenizing a string at first glance only requires to split the string on every whitespace and newline. Source code, however, has some quirks that have to be additionally fil-

Figure 2.1: Example for an AST[24]

tered out. For example, the basic command to write a string to the command prompt in Java is "System.out.println("string")". At first glance, it might appear to be a good idea to not split that at all, because it is usually used in that exact composition. Though one could for whatever reason save System.out to a local variable and call it foo. Then the equivalent call would be "foo.println("string")". Splitting the string would result in a similarity of 2, matching "println" and "string", with the highest possible similarity being the number of tokens of the smaller of the compared token sets, which is 3 here. If one were to not split the statement, the similarity would be 0, with a maximum possible similarity of 1. Then there is the following case:
a=b+c compared to a = b + c
This would have a similarity of 0 if "+" and "=" were not listed as noise characters, because the first example would be just one token and the other would have five single-

char tokens. One could argue, that those are, albeit their length of just one character, full statements and should count as tokens themselves. They are also very common statements, and just like "this", "and" and "that" in natural language processing, they can be filtered out as insignificant.
Then there are phrases that are filtered out as noise characters like "+@+" and "@@::@@", though they should be quite rare in code files. Those phrases are used as separation flags in the digestion process of the CCD and would break the format if not filtered out.

2.7 Code Clone Detection

This is the aforementioned other heart piece of the study. One main concern in this thesis is checking source code for malicious/vulnerable code. This is called (security) code clone detection. It means, that examples for various vulnerabilities are compared to other code, to see, if they match in any way, to identify vulnerable code in an automated manner.

2.7.1 Clone Types

Various metrics have been declared concerning the topic of code clone detection. Among those is the metric of clone types.[7] These clone types are hierarchic from the loosest type (type 4) up to the most narrow (type 1). Here, code clones are classified in certain bounds of similarity as follows:

Type 1

A clone of this type is an exact match to the referenced code. Allowed differences are whitespaces and comments. It has high precision and low recall.

Type 2

This type is almost like type 1, except here, changed variable names are allowed. Implementing a CCD capable of detecting type 2 clones, without also being able to detect type 3 clones as well, is quite difficult. This is due to the contextual information needed for reliable type 2 detection being already sufficient for type 3 comparisons.

Type 3

This type is the first to prove very difficult to detect because it is the first one to allow changed or added statements. This means, that a dumb comparing-algorithm will

match code with an irrelevant added statement in it as type 3 — correctly — but also match code with significant statements added — wrongly.

Type 4

These matches are semantically identical but can differ largely in syntax. Detecting these in an automated manner proves very challenging because even an advanced detection like an AST-based CCD might not show a match for two functions that are written in completely different ways but do the same thing.

The types 1 and 2 are fairly trivially detected in most CCDs, which makes types 3 and 4 much more interesting. Though that is not the ambition of this thesis, for here an existing CCD is used, instead of creating a custom one. This existing CCD is a token-based CCD, which, according to G. Kaur and S. Sharma[7], is by definition unable to detect type 3 and 4 clones.

2.7.2 Variants of Code Clone Detectors

There are several approaches to code clone detection[7].
The text-based method is only capable of detecting type 1 clones because entire code segments are compared without any flexibility. It takes some chosen measure of text segment, which usually is just a line of code, and does a text search in the source code in question. This can practically be done by hashing the entire segment and searching for the hash instead, greatly improving the performance as opposed to full-text searches. This rules out the detection of type 2 clones because different names for variables are not recognized.
The token-based method, which is used in the software of this thesis separates the text into - usually - per-word n-grams. Higher n can result in higher complexity but also higher precision - in this thesis, n is 1. CCDs of this type are incapable of intentionally identifying type 3 clones since in those, the code has been altered with removed or added statements and that would only reduce the similarity-rating of the comparison.
As an example, here is a sentence with its tokens:

This is a sentence, a sentence should sentence with a prison sentence.
$[this : 1; is : 1; a : 3; sentence : 4; should : 1; with : 1; prison : 1]$

Here it becomes clear, that without further meta-information about the tokens, this is even unable to reliably identify type 2 clones. The three different meanings of "sentence" are ignored in the digestion of this example, just like a token-based CCD ignores the meanings of its code tokens. In total, the certainty to have found a type 1 clone provided by a text-based CCD is sacrificed in favor of a probabilistic detection of up to type 3 clones without certainty for any type.

The AST-based method utilizes any way to partially compile the code in question. Usually, that means that a piece of code mimicking a compiler for the respective language generates the AST from both codes, comparing their nodes and edges. This method can easily detect type 1 and 2 clones and even type 3 should be no issue, while also being able to identify the type of the clone, too. Unfortunately, this third method is still unable to detect the fourth type of clones.

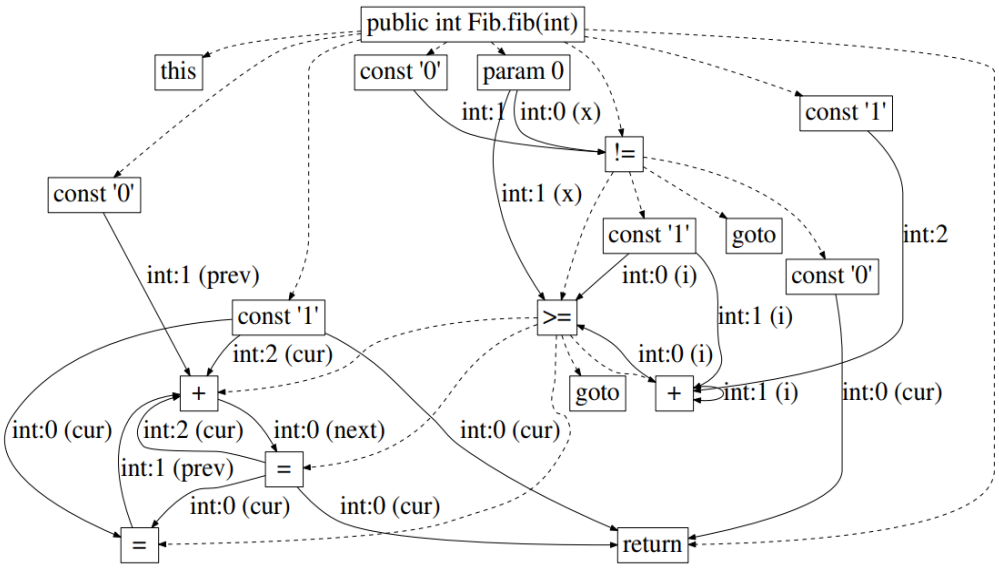

Figure 2.2: Example for a PDG[1]

To detect type 4 clones, code has to be converted into a so-called Program Dependence Graph[1] (PDG), for which an example can be seen in Figure 2.2. This PDG shows all data dependencies and control dependencies in the code. A PDG-based detector is very costly, and its results are not unimpeachable either, as they are often not confirmable by humans and do not have a precision of 100%[1].
Two other common methods for CCD exist. One of which is a metric based approach. In it, the code is measured using chosen metrics like Line(s) of Code (LOC) and code repetitions. The other approach is combining at least two of the approaches, hence its name, hybrid method.

2.7.3 Security Code Repository

This is a repository hosted by the Software Engineering (SE) group of the Leibniz University in Hannover[30]. It contains various code examples for vulnerable code written

in, for now, Java, JavaScript, and Python. For each example, there is also metadata added, such as CWE-id and source. This repository is solely important to the CCD and its validity is discussed in the future work chapter. It is automatically being cloned and kept up-to-date by the software of this thesis.

Extraction Tools

Two tools have been created prior to this thesis to ease the manual creation of security code examples.
One focuses on searching for security code examples on GitHub.com and is called the SecurityCodeExporterForGitHub[31]. For that, it just uses the "CVE-<YEAR>" part of the keys as search keywords, inserting every year from the start of CVE collection until the current one. It then creates a database consisting of all the search results for each word, plus extensive metadata for each result, such as URLs to the repository, the commit, the profile carrying out the commit and more. The other tool, the SecurityCodeExporterForStackOverflow[32], works similarly. It searches for examples on (by default) StackOverflow.com[46], security.stackexchange.com[47] and codereview.stackexchange.com[50] using a manually assembled list of keywords. The results of this tool contain URLs to the respective posts and some processed contents of it, including the author, a link to his/her profile and its reputation rating. The result set of these two tools mainly differs in that the GitHub exporter searches for CVE keys instead of keywords. If it searches for "CVE-2004" and receives a result, it will usually be, because the commit message contained for example "CVE-2004-1134", which is then extracted and saved into the result set. The SecurityCodeExporterForStackOverflow demands the manual insertion of CWE identifications, instead. Here, these CWE are not associated with CVE or CVSS data, so results from this exporter cannot be analyzed via CVSS. If the StackOverflow security code examples were to be assigned to CVE-ids, one would have to assign every CVE-id of the assigned CWE to it, leading to a large collection of CVSS ratings, which could only be used to calculate statistics about the particular CWE, serving no purpose for a particular code block.

Security Code Repo Tooling

This is a tool created to add to the extractor tools and assemble their results into one. The extractor tools create databases containing all matches for the specified search criteria e.g., vulnerabilities in Python. The tooling uses those databases to assist a reviser in manually working out the examples used in the CCD. It creates a dataset consisting of the given metadata of the example -like a link to the GitHub.com commit or the StackExchange.com topic), a manually assigned CWE classification in case of the StackOverflow dataset, and a manually extracted code example from the source for the tokenizer to use.

2.7.4 SourcererCC

The SourcererCC[8] is a token-based CCD that is utilizing multiple phases of filtering and an inverse index for runtime optimization. It does not do the tokenizing but instead requires the tokens to be prepared by separate means, such as our modification of an ANTLR-parser for Python. The process of the detection including the tokenization goes as follows:
First, the code needs to be parsed in a way that is capable of splitting up source files into code blocks of the chosen granularity. If one were, for example, to pick the token granularity to be method based, the source code would have to be parsed to recognize such bounds. In the software created for this study that is done for Python using ANTLR. The tokens created from the code blocks distinguished by ANTLR are then inserted into the SourcererCC algorithm that starts by indexing the tokens to improve performance. The details of the token processing are thoroughly described in the paper of the SourcererCC, thus it will only be described in shallow detail, here. In short, when two code blocks are compared, the union of their token sets is calculated. When both sets contain a shared token, both sets contain a frequency for it. The lowest of these frequencies is the similarity rating for that token. This similarity rating is then summed up for all tokens in the token set intersection. It is practically a counter for common tokens and can then be combined with the total token count of both sets, to decide if a code block will be classified as a clone. For example, in the paper of the SourcererCC, a code clone candidate is accepted as code clone if:

$$similarity \geq \lceil \theta \times max(|tokenSetA|, |tokenSetB|) \rceil$$

Here, $|tokenSetA|$ and $|tokenSetB|$ are the total token counts of the code blocks' sets and θ is a threshold factor that has to be chosen beforehand. In the paper of SourcererCC, θ is chosen to be 0,8, apparently without explanation. In this thesis, θ will be calculated using existing results. This is discussed in chapter 3.4. This one-dimensional rating of similarity cannot be used to identify what clone type has been encountered.

3 Concept of the Tool

In this chapter, the software will be outlined, describing the modules it traverses, from start to end. Keep in mind, that the codebase already contained CCD and LC for Java and JavaScript, so the majority of explanations and examples will regard the Python addition done throughout this thesis.

3.1 NVD Dump

The first thing for the tool to do, is creating a local copy of the National Vulnerability Database (NVD) because National Institute of Standards and Technology[26] (NIST) does not provide direct access to its Database (DB). The database is offered as a feed on the NIST website, in the form of JSON files. These either represent all CVE of one year per file or a "modified" file, which contains all additions and modifications to any CVE done within the last 8 days. To enable our tool to utilize this data source in the LC, as well as the CCD, the NVD Dump is created from these feeds. For the dump, all feeds are downloaded, parsed and inserted into a single database.

3.2 Git Crawler

The Git Crawler is a piece of code used to iterate over GitHub.com repositories using their API. It alternates between the implemented languages — Java, JavaScript, and Python — to start the analysis process for repositories marked as using those languages. The processes explained in the following three sections are repeated for each commit of each repository. Here, the GitHub.com-rate-limit is handled, as it will only allow ten repositories to be queried via the API every minute. So if need be, the Git Crawler will notice, that the rate-limit has been reached, retrieve the reset timestamp in epoch format, and wait until it can do more queries.

3.3 Library Checker

The LC has two jobs. First, it has to retrieve the libraries used in the repositories. This is language-dependent and each language has individual code to find the files containing the library identifications. This is the first process per commit, though, since the three

main modules of the tool are independent, the order does not matter. After it retrieved those files and thus, extracted the names of the libraries used, it searches the NVD dump for CPE representing these libraries. Then, it will search the NVD dump for all CVE entries referencing the matched CPE entries. The CPE repertoire of the NVD does not only contain CPE entries referenced in CVE entries. Some are referenced by no CVE entries. In the local dump though, only the CPE entries referenced by CVE entries are saved. This means, should a CPE match be found for a library, a CVE would be found as well. If there is a CPE match, the library and all its associated CPE entries and CVE entries are saved into a collection within an object gathering all results for one commit. As an example, here are the first four rows of the "requirements.txt" from the Python test folder used in the evaluation of the LC.

numpy
thislibrarydoesntexist
theano
numpy==1.8.1

This is the dependency management that the Python LC hooks into[53] and it is quite simple, as each line contains one library. Optionally, it also contains a relational operator and a version identifier as can be seen on the fourth line. This version identifier does not have to be a number. This particular system will never contain a vendor identifier that could be used in the query for a CPE match, but the library names tend to be sufficient for precise identification. The LC result set for this requirements.txt contains "numpy" and "numpy==1.8.1", as "thislibrarydoesntexist" is a fictional library, does not exist and thus should not appear in the NVD. "theano" is an existing library but is referenced in no reports in the NVD. The result for "numpy" contains two combinations of CPE and CVE:

CPE	referencing CVE
cpe:2.3:a:numpy:numpy:*:*:*:*:*:*:*:*	CVE-2019-6446
cpe:2.3:a:numpy:numpy:1.8.1:rc1:*:*:*:*:*:*	CVE-2014-1859

The result for "numpy==1.8.1" is only the latter. The query will get as fuzzy as necessary. If there was no result for "numpy==1.8.1" it would have looked for numpy only and treat all results as matches, ignoring the version stated, since they are not entirely accurate anyway as stated by Knorr et al.[70]. As for the consideration of the choice of the relational operator, should a "!=" (not equal) be used, the library would be treated as if the line was not in the file, whereas all other operators would be treated the same as "==". The test-projects are not separated into commits, and thus the whole test project is treated as a single commit and both these libraries and their matching NVD excerpts will be saved to the same result set. The situation, where a matching CPE key is found

with no CVE linking it, would be possible in the real NVD. Here, in the local dump, only CPE entries that are referenced by CVE entries are persisted, so that situation will not occur.

3.4 Code Clone Detector

After the Library Checker[21] (LC) is done, the CCD starts. This first filters the files by ending, for example, ".py" is the only ending allowed into the Python-specific CCD. This is done for all languages, ignoring that a repository was selected based on a specific language. Most times the outcome is, that if a repository was started for one language, the filter will only encounter files for that same language. Sometimes though, repositories contain source files of multiple languages and while the repository is predominantly written in one language, it might still contain vulnerabilities that need to be detected by another language's CCD processes. After the files have been sorted by language, the files are read, deconstructed into the desired code granularity, and then tokenized. The tokens are the input format for the CCD that was used in this study - the SourcererCC[8]. The SourcererCC will generate a similarity rating for all combinations of reference repository token sets and input repository token sets.
The first of three clone candidate filtering phases is that all results with less than two common tokens are removed from the set. These code clone candidates are then enriched with the meta-information of the reference repository.
One of the reference repository's code examples is this:

tempfile.mktemp()

This is the entire file. Notice how this does not contain a compound statement but is only a single global statement. This is why a special case has been inserted into the Python tokenizer. The granularity used in the Python CCD is aimed to be at "compound statement" level, which matches next to every kind of grouping, like "if", "for", "while", "try", "switch" and more. Should a file not contain such a compound statement, the granularity is raised to file-level, which means the file is tokenized as a whole. Another peculiarity of this example is, that it will as a whole result a total of two tokens. With the CCD used, this means it cannot yield a similarity greater than two.
The second filter used in this study calculates a relative similarity by dividing the similarity by the reference repository's token set cardinality. For the example given, this would mean the relative similarity is either 0%, 50% or 100%. At that point, the minimal relative similarity needed for the code clone to be considered as such is chosen to be 50%. This is where a difference in the concept of this study to the SourcererCC paper comes into play. In the paper for the SourcererCC, clones are only accepted as such if they exceed the bigger cardinality of the compared token sets, times a threshold factor of 80%. This means, should these two tokens be found in a code block with one hundred tokens the match would not be considered a match.

After this second filter, a third one is applied. This third filter is based upon the filter used in the SourcererCC paper. It uses the same formula, but θ is calculated using the results from the first two filter phases. The idea is, to not filter too rigorously, so that it is ensured, that there are discussable results.

There were three approaches to calculating θ. The first was to take the nth code clone candidate from a list that is sorted by a value that was calculated by solving the formula from the filter for θ:

$$matches \geq \theta \cdot max(|tokenSetA|, |tokenSetB|)$$

$$\theta = \frac{matches}{max(|tokenSetA|,|tokenSetB|)}$$

$$n = |commits|$$

This would lead to one clone candidate allowed per commit processed. Then, the resulting θ would be averaged over all repositories, before really being applied. On one hand, this lead to errors, because some repositories did not have more clone candidates than commits, on the other hand, this resulted in a really low value for θ (around 20%). To fight the issue, where repositories could sometimes have fewer clone candidates than commits, the second approach was chosen:

$$n = \frac{min(|commits|,|candidates|)}{2}$$

This mitigated the first problem, but still yielded a very low θ of about 30%. The third approach is the one that was eventually used:

$$n = \sqrt[2]{min(|commits|, |candidates|)}$$

Averaged over all repositories, excluding repositories without any candidates, this resulted in a θ of 48%.

3.5 Keyword detection

In addition to the CCD and LC, the commit messages are scanned for matches with a predesigned list of keywords. This might give some more insight for found vulnerabilities, should a relation between keywords and CVE or CWE emerge.

Choice of Keywords

The keywords used for the StackOverflow exporter tool have been taken from this paper[51] by A. Bosu et al. and the same keywords have also been used in the keyword detection. They are grouped by CWE. Some of those keywords were just too vague and yielded a very high recall with pretty low precision and have thus been sorted out. Also, using compound keywords consisting of multiple space-separated words simply creates a search for either, so, if "gain access" is a useful keyword, its usefulness is reduced to "gain" and "access" somewhere in the result. Not all categories fit the theme of this study, as it is aimed at security-related vulnerabilities and "Integer Overflow" is none - at least for Python code. Aside from the keywords used in the referenced paper, a few NVD-related keywords have been used. The keywords used for the search to assemble the reference repository have also been used to create a list for each scanned commit. This list will contain all keywords that have been found in the commit message and will be present in the result set of the respective commit.

3.6 Databases

Five databases are currently embedded in the tool. One database for each language, currently three total, and easily extensible for more languages in future studies - their entity-relationship diagrams will be omitted here because they were not designed in the course of this thesis. One database roughly mirroring the NVD, and one, where the result sets are deposited. Both are outlined in the following entity-relationship diagrams.

NVD Dump

As can be seen in figure 3.1, the "cpe" table has been amended by four hash columns, to avoid querying for strings and allow indexing, which used to be a large bottleneck in the LC implementation. There is also the meta table containing a timestamp. This is used to persist the time of the previously performed update to decide if either a full dump needs to be pulled when the timestamp is older than eight days, or else, just the "modified"-file needs to be pulled.

Tool Results

This schema depicted in figure 3.2 does contain some clutter and redundancy, but since it will be refined further, at this point as much data as available is saved. It is simpler than it looks. The "result per commit"-table links five tables, of which two contain more metadata about the commit itself. The other three are the result sets of each processing

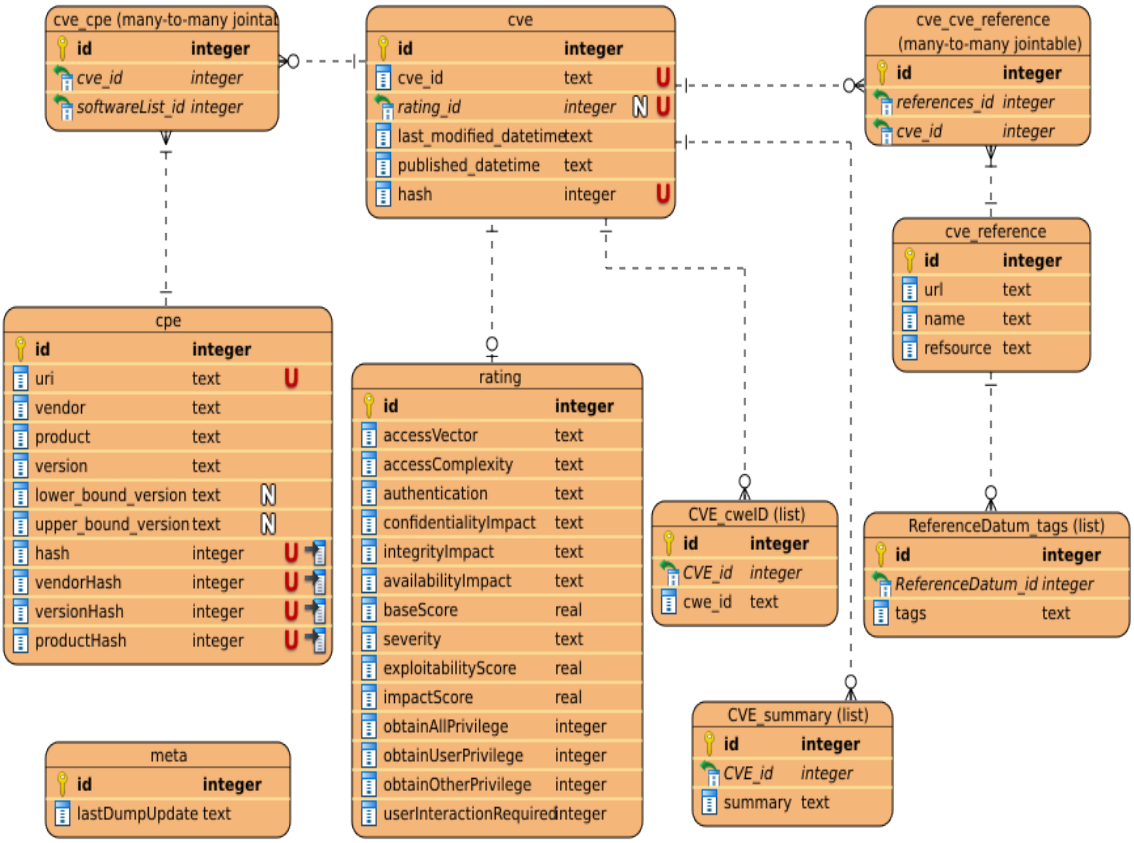

Figure 3.1: Entity-Relationship-Diagaram for the NVD dump

step: LC, CCD, and keyword detection. They are completely separate from there, except for the "cve"-table, which is used in both the LC and CCD result sets.

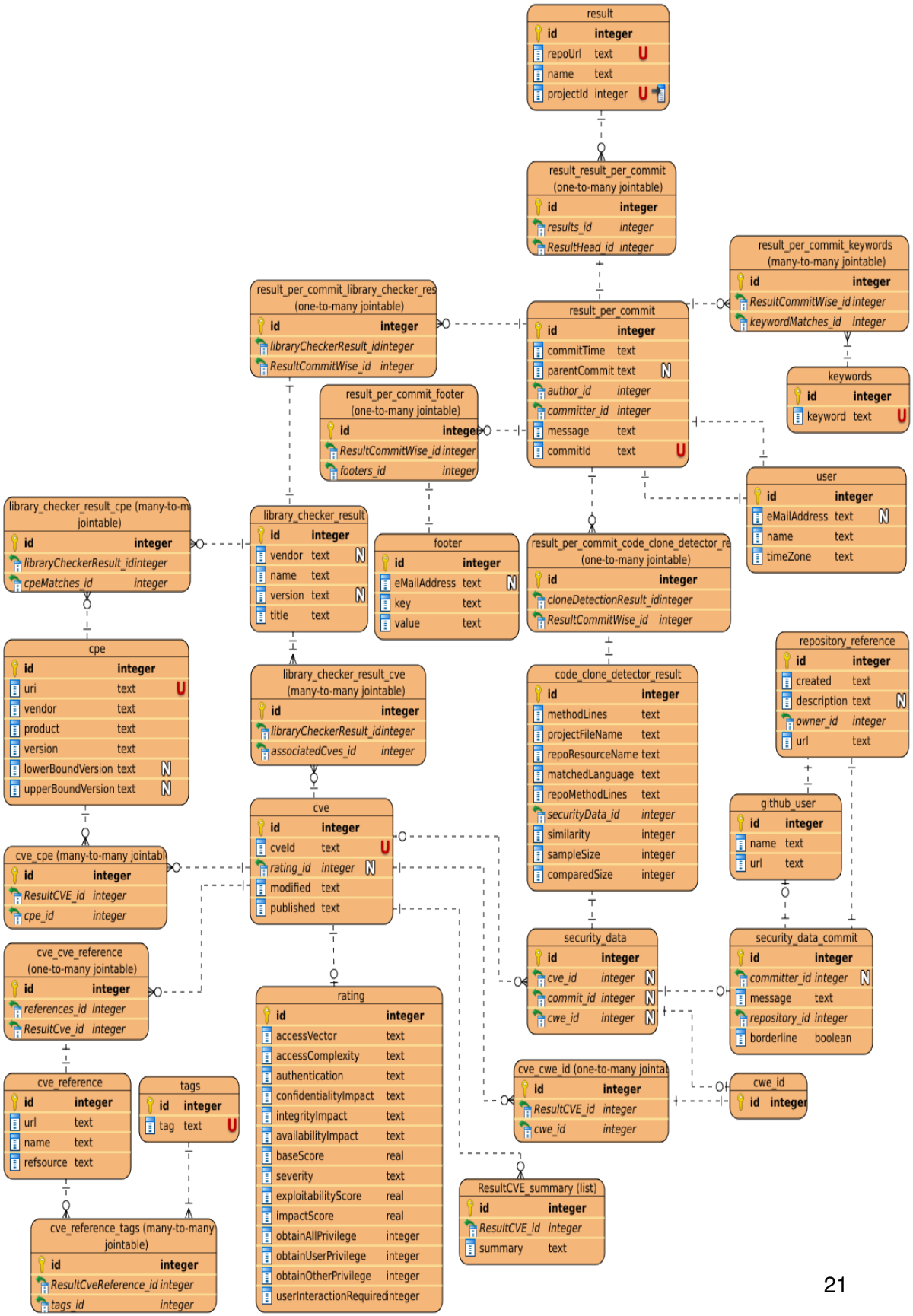

Figure 3.2: Entity-Relationship-Diagaram for the result DB

4 Implementation

In this chapter, the tools, libraries and similar, used in the software of the thesis, will be described.

4.1 Initial Code Base

Our tool was not built from scratch. It is based upon a plugin that was written for JIRA[56], meant to analyze repositories managed over it. Most of the pieces taken from that codebase, have been changed or removed, but worked as a semantic guideline, for the software's skeleton. The tokenizers and LC of Java and JavaScript have not been changed. SourcererCC, which was not inserted via a library, but as plain code instead, has been changed a bit to not conflict with the threading. Like before, the utilized database software is SQLite[58], though the ways in that it is used have been replaced entirely. All remnants of JIRA have been removed.

4.2 SQLite

"SQLite is a C-language library that implements a small, fast, self-contained, high-reliability, full-featured, SQL database engine."[58] It has been used here for three reasons. First, it was used in the initial codebase. Second, it is a system that is fully contained within the tool. If, for example, MySQL[68] had been used, the executing system would have needed MySQL on it, as it is designed for immobile, stationary databases, whereas SQLite works with local files, that one can conveniently move around. This comes with a tradeoff, as, since SQLite is designed for more volatile databases, it is mostly incapable of altering already written data and parallelization is barely possible. Third, all related tools work with SQLite. This means that even if this one were to use MySQL instead, it would still need SQLite to read the external databases.

4.3 Hibernate

Hibernate[55] is an object-relational mapping tool for Java. It provides a framework that is used to map Java objects to relational database entities as seen in figure 4.1. If

the objects of the Java code match the tables created in the SQL code, one could just persist one complex object as a whole.

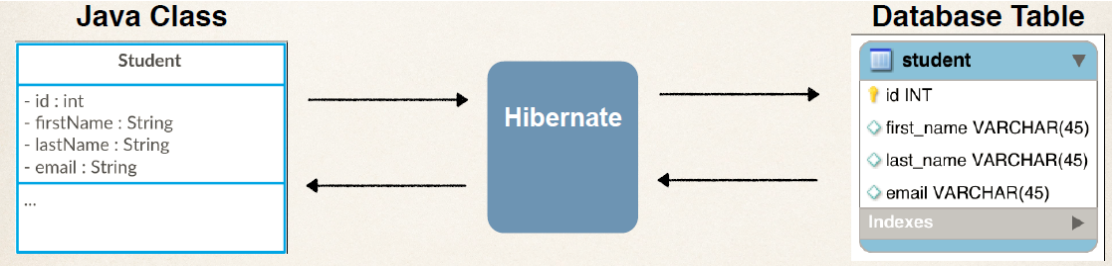

Figure 4.1: Depiction of the Hibernate functionality[57]

The only time, this "Hibernate" software is used directly is the instantiation of each of the used "dataSources", as they are called in Spring Data. All else is done indirectly via Spring Data.

4.4 Spring

Spring[63] is an open-source Java framework meant to ease development with Java and Java EE[64] and is composed of many modules. Spring Boot[65] is used to set software up for tidy code and prints, and to build a base the other Spring modules can be hooked into. With Spring Boot, lots of annotations can be used, such as those declaring classes as Spring components. Should, for example, a class be declared a Spring service - which is also a Spring component -, it would be noted in the "component-scan" that Spring performs at the start of one's application. All those components are put into a dependency tree and if needed, so-called beans are generated for each of them, which could be compared to static variables in the case of Spring. If class A is declared as service and then used in class B via dependency injection, Spring calls the bean to create an object of class A to inject it into class B. If there was then a class C that also contains an injected field of class A, the same object spring generated for class B would be passed to C, too. This, combined with the later-described Lombok[61] library, makes for very tidy and organized code. Also, Spring has a built-in configuration file system. By default, it searches within a declared "resources"-directory for a file called application.properties, in which you can define some Spring internal variables, such as logging verbosity level, but also custom ones that can be used akin to dependency injection via beans. This is used to some extent in the software of the study, as the thread limit, the target number of repositories to be scanned, and the URL of the reference repository are all managed via such files. Another Spring module used in this case is Spring Data[66]. It is a library, based on a Java Persistence API[29] (JPA)

implementation of choice, like Hibernate[55] - in our case - or Eclipse Link[67]. It is used to access those JPA implementations in a more abstract and unified way. Once you have configured your Spring Data "dataSource" with all entities, databases and JPA implementations, access to those "dataSources" is mostly reduced to annotations and generated queries.

4.5 ANTLR

ANother Tool for Language Recognition[10] (ANTLR) is currently not directly used within the software, but it is a vital component of it. The current utilization of the ANTLR software is to manually execute the ".jar"-file with a manually picked ".g4"-file as input, to have it generate a language parser from it. This parser is then put into the code base and dumbed down to a tokenizer. There is a plugin for Maven[54], that could be used in the future, to automatically parse all languages that are available in the official ANTLR GitHub.com repository[9], extending the CCD capabilities of the tool to hundreds of languages.

4.6 Maven

"Maven is a software project management and comprehension tool."[35] It is used to manage project structure, jar building, dependencies and more. Maven is coupled with a website[62] providing online sources for most libraries one could need. Once a dependency is added to the central Maven configuration file of the software, called "pom.xml", Maven downloads the corresponding library, which is then immediately ready for use in the software. Maven is one of those systems, the LC could get its library references from.

4.7 JGit

JGit[59] is a library maintained by "The Eclipse Foundation"[60], implementing the Git[34] functionality in Java. In this project, it has been serving to map a combination of local folders, repository URLs and Java objects together, and to manage most Git interactions like cloning, pulling and retrieving difference-reports between commits. It will, or rather should, ultimately be removed entirely, though. That is because it is prone to various errors that do not originate from Git, and its functionality can easily be replaced with integrated command line calls, though that would introduce the requirement of Git being installed on the executing system unless Git can be included into a "jar"-file as well.

4.8 Threading

Threading has been a big topic regarding this thesis' software because most of the codebase provided by the underlying JIRA[56] plugin did not support threading, though, at the first glance, it appeared to already be using threading. Because of this, the plugin was not only converted into a standalone tool but equipped with "embarrassingly parallel" threading.

The number of threads utilized is 2+n, where n is the number specified in the application.properties. The first thread is the main thread and takes place only within the AnalysisManager class. It handles the initialization of all databases and the reference repository and its tokens, then starts the second thread and from then on handles the persisting of result sets into the result database.

The second thread is the Git Crawler. It iterates through a list of repositories for each language via the GitHub.com API[11], assembles n AnalysisThread instances and starts them. It will do so until it either hits the defined limit for AnalysisThread instances - which is set in the "application.properties"-file - or the GitHub.com rate-limit[52] for API usage, which limits it to ten "queries" per minute. If it hits either of those bounds, it will wait a few seconds and try again until the defined number of repositories per language - again, from the "application.properties"-file - is reached and then wait for all AnalysisThread instances to finish.

The other n threads are the AnalysisThread instances. They represent the extent of one repository of GitHub.com each. These repositories differ largely in size, and while some threads will finish within seconds, making room for new ones to start, other threads can take hours, days, or even weeks to finish. The thread will iterate over all commits of the main branch of the repository from newest to oldest, always extracting the difference of the current commit to the previous commit. This difference is then used in the three processing stages, LC, CCD and keyword detection, of which each could have been a thread instead, because, in the current implementation, they are mostly independent. This separation would have made sense in the current implementation, but not with future work in mind, as the possibility of dynamic ANTLR integration and hundreds of CCD languages would need the LC to be remotely as extendible. To achieve this extendibility, the one implementation coming to mind, would be the LC using libraries extracted by ANTLR and thus, the LC depending on the CCD. Each of these 3 processes adds their result to the wrapping object, the CommitWiseResult, which is, as the name indicates, instantiated for each processed commit. Each CommitWiseResult object is added to the ResultHead of the thread at the end of the commit's processing, even if neither LC, CCD nor keyword detection had any results. Next to the result data, it contains data about the commit and its parent commit. This allows connecting the CommitWiseResult objects into a chain, possibly revealing more information about the handling of vulnerabilities. At the end of an AnalysisThread, its ResultHead object is passed to the main thread, which will then save it to the result database in a single transaction.

The verbosity of the tool has been tuned down a lot, so it can happen, that even with 32 threads there won't be a single print for hours. That is why in every thread except the Git Crawler, optional hourly prints have been added, just to be sure the tool did not freeze - which it is not expected to do.

While most aspects of the threads are entirely separable, a few are not. The reference repository tokens and the NVD dump have to be accessible by every thread, but since these are not modified after initialization and are only read afterward, they are thread-safe. This level of thread independence comes with a great benefit because it is scalable to multiple cores almost linearly. With $n = 32$ used in the process of generating the study's data and one thread surpassing the limit through a tolerated race condition, resulting in a total number of 35 threads, the CPU utilization shown by the "top"-command[69] on the Linux shell showed to be around 3045%. That is 87% uptime per core. And since two of those threads spend most of their time waiting, the utilization of the others is likely even higher.

4.9 Lombok

"Project Lombok"[61] is a library used to generate - and thus, hide - code that would otherwise cause massive clutter. For example, a wrapper class containing 18 fields, including both primitive and complex types, could normally easily span over 200 LOC, of which about 30 are of any substance, while the other 170 are constructor, setter, getter, and other trivial code. With Lombok in use, one could annotate the class with "@Getter" and "@AllArgsConstructor" and others like that to generate this trivial code. A class of 400 LOC - this happened to me in this study - can shrink down to 40 Line(s) of Code (LOC), by letting Lombok generate logger, getter, setter, multiple constructors, "toString"-methods, "equals"-methods and hash functions with only seven LOC. This, of course, is not limited to class-wide generation and can, if needed, be used with more precision. To inform Integrated Developer Environments[27] (IDEs) about the presence of the code that is only present after compilation, a Lombok plugin for the respective IDE - most common IDEs are covered - is required.

5 Results and Discussion

This chapter is about the results produced by our tool. In the first section, the Python-related security code clone detection and library checking will be evaluated using a test Project. In the section "Sample Repositories", four repositories are discussed. One repository for each of our languages, and one repository containing multiple languages, have been picked and discussed, to show the extent of information gathered for each of them. After that, in the section "Compiled Results", the combined result set of all scanned repositories and the research questions will be discussed. In the last section of this chapter, "Threats to Validity", weaknesses of the concept of each scanning module - keyword checking, library checking, and security code clone detection - will be discussed.

5.1 Evaluation of Tool (Python only)

The Python-related parts of the tool have been evaluated, while the other two languages were assumed to be evaluated from before the start of the thesis.

Security Code Clone Detection

For the CCD, examples for types 1, 2 and 3 have been manually created.

For type 1 clones, that means the reference repository example code files have been copied to the test-project. For type 2 clones, the code is copied again, but variable names have been changed randomly, keeping in mind, that variables with multiple occurrences will undergo the same changes within a code block. Variable names that are assumed to come from libraries have not been touched, so only variable names that have been chosen within the context of the example have been altered. For type 3 clones, a single line has been added to most of the test files created for type 2, while in a few, random lines have been removed instead. In some cases, the files were unfit to be altered into the next type of code clone. For example, files with a total token count of 2 have not been used in the type 3 test creation, as removing statements would lower the token count below the minimal similarity of 2, and adding statements would overtake the total token count too easily.

The tool has then been executed with the created test files as input. The results produced by the software have been evaluated, as seen in table 5.1.

True positives are the matches where both, test file and security code example, share the same CWE-id.
The maximum number of true positives produced is calculated like this:

$$max|truePositives| = \sum_{i}^{N} |exampleBlocksForCWE_i|^2$$

Here, N is the set of all CWE-ids referenced by the language's security code examples, and $exampleBlocksForCWE_i$ is the number of code blocks for each of those CWE-ids. Remember, that one file can contain multiple code blocks. The maximum possible precision would be if all matches were true positives.
The maximum possible recall would mean, that every single code block produced a match with every single code block associated with the same CWE-id.
The maximum number of matches can be calculated like this:

$$max|matches| = testFilesCodeBlocks \cdot exampleFilesCodeBlocks$$

And from that we can deduce the maximum number of false positives:

$$max|matches| - max|truePositives|$$

False positives are the matches that connect examples of differing CWE-ids. Unmatched files are test files that did not match at all. The test files did not include non-vulnerable code to test for false positives. Instead, the difference between vulnerable and non-vulnerable code is treated the same as the difference between one CWE-example and another because their examples do not contain each others' weaknesses.

	type 1	type 2	type 3
example files	47	36	39
code blocks	51	36	43
max |matches|	2.601	1.836	2.193
max |truePositives|	653	548	587
matches	58	44	31
unmatched files	0	0	12
true positives	58	44	31
recall	8,88%	8,03%	5,28%
precision	100%	100%	100%
F1 measure	16,32%	14,86%	10,03%

Table 5.1: F1 Measure for the Python CCD

Recall, precision and F1 measure are calculated like this:

$$recall = \frac{|truePositives|}{max|truePositives|}$$

$$precision = \frac{|truePositives|}{|matches|}$$

$$F1\ measure = 2 \cdot \frac{recall \cdot precision}{(recall + precision)} = 2 \cdot \frac{|truePositives|}{max|truePositives| + |matches|}$$

The above table is basing its measures on the assumption that with a recall of 100%, one example would be matching every single test file with the same CWE-id, resulting in quite low numbers. If instead, the question was, "has every example of the tests been found by any of its CWE-id matches", the answer would be yes for type 1 and type 2 and no for type 3 because in its result-set 12 files were missing. The reason for that is, that the changes made to those files were too large in relation to their initial token count, which made them drop below the 50% threshold. None of the examples contained matches that were connecting different CWE-ids and thus, "false positives" have been zero for all three tests. Of the matches between the reference repository and the test files created from them, the similarities of each file and it's descendants have been collected in the following table 5.1.

type	max similarity	min similarity	average similarity
1	100%	100%	100%
2	94%	50%	78,09%
3	81%	50%	61,9%

As expected, for type 1 clones, similarity is always 100%, because the files were practically only compared to themselves. The minimal similarity of type 2 and type 3 is 50%, showing that - would the threshold be a little higher - not all test files would find a match for type 2 and even less would match for type 3. The average similarity is decreasing from 100% at type 1, down to 61,9% at type 3. This number would be way lower if matches that have been sorted out by the 50% threshold had been considered. This occurrence really being connected to the clone type is unlikely, as the CCD cannot identify the nature of differences in the tokens. The capability of the CCD to detect type 2 and 3 clones depends entirely on the degree of changes, instead of just their type.
Consider an example file with 8 total and 5 distinct tokens, with one token occurring 4 times. If one were to change that one token by a single letter, the relative similarity would drop to 50%. If instead, one were to change one of the other four tokens, the relative similarity would be 87,5% for the same type of clone for the same example.

If a calculated minimum relative similarity is desired, one would have to consider statical values for the number of tokens per code block, the number of reuses of the same variable, a distinction of chosen and preset variable names and possibly more values that are hard or impossible to procure. The SourcererCC does not provide such, but a separate similarity value only showing the distinct matches, without including their frequency would potentially make type 2 detection a lot more precise, as can be shown using the example introduced above: A file with 8 tokens, 5 of them distinct. One with a

frequency of 4. That one is changed. The resulting similarity is 50%, while the resulting "distinct-token-similarity" would be 80%.

Library Checker

For the LC, a "requirements.txt" was created, containing the names of libraries, of which most are expected to be found in the NVD. A total of 80 libraries were put into the "requirements.txt", with the first 4 of them accounted for in the "Concept of the Tool"-chapter. The other 76 are all libraries that occurred in various repositories and are confirmed to have entries within the NVD. They are all expected to produce matches. The libraries are not required to be actual Python libraries, as the related language is irrelevant in the CPE dataset as well. Of the 80 libraries, numpy, in particular, appears thrice. Once with a version specified, and twice without. This duplicate occurrence did not appear in the result set. The total number of libraries, for which matches have been found is 77. The three missing results were the duplicate numpy, the imaginary library "thislibrarydoesntexist" and the library "theano" that has not had results in the NVD, using the NVD's own search function either. The result is, that of 80 libraries, where 77 were expected to match, 77 did indeed match. That means, that the precision and recall are both a hundred percent for that matter.

Version identifications are not meant to be precisely detected by this tool. With the hierarchical queries done to find matches for libraries, explained in section 3.3, the matches for libraries with specified versions would have to be filtered again. With these hierarchical queries, a recall of 100% is ensured, so filtering afterward will not lack any results - but this thesis will not involve such filtering.

5.2 Sample Repositories

After letting the tool run until it persisted 159 processed repositories' result sets, the result-DB contained 400.268 total commits, 2.533,34 average per repository, excluding one repository that had no commits, because the author was forced to delete all of them. The repositories were processed prioritized via the number of GitHub "stars". Of the list of repositories on GitHub.com that has been used, not all repositories will be found in the result-DB, as the processing of the repositories is very time consuming and will not finish in the same order it was started in. In fact, a few repositories had such a large commit history, that the tool did not finish processing them in ten days. Means to improve performance will be discussed in chapter 8, "Future Work". Also, the parsing can fail for single repositories, if an exception occurs. In that case, the thread in which the exception occurred is the only one that stops. Currently, there is one known exception thrown by JGit which aborted the processing of a single repository.

Security Code Clone Detection

For all the commits, 417.355 CCD results were persisted, including non-matches. Each CCD result contained either a match or a non-match, as files without matches have been persisted as well. This is used to identify fixing commits. 229.962 of the results are non-matches and 187.393 entries with any kind of match, filtered using the third filter explained in section 3.4.

The following numbers are all based on the filtered CCD results. Files that match multiple security code examples have one entry for each of those matches. 282.649 distinct files have been scanned. A total of 140.182 distinct files have been matched with any security code example, whereas 142.467 files never produced any match. On average over all repositories, every file matched 0,66 times and did not match in 0,81 commits - not accounting for files that had a match in every single occurrence. Averaging the match-frequency only over files that ever had a match, raises the average matches per file to 1,34.

Library Checker

In all the commits, a total of 278.103 LC results were persisted. The number of commits that contained any LC results is 64.764, which is an average of 4,29 results per commit. 16,18% of all scanned commits contained matches from the LC.

Four Sample Repositories

To show the generated data in detail, the results of four repositories will be explained in particular. One for each language, containing only matches for that language - no mixing at all - and one repository containing matches for two languages. There were repositories with files for all three languages, but after applying the third filter to the CCD results, no repository contained matches for every language. For the example repositories, the ones closest to the average commit count are selected. All data comes exclusively from the result DB unless stated differently.

5.2.1 Java Repository

The chosen repository is the one with the id 66, containing 2.457 commits. It is called glide[73] and was created by Sam Judd (sam@bu.mp) on December 20th, 2012. A total of 162 users have contributed to the repository so far, 112 of those made only one commit each. 8 of the 162 users appear to be alternate accounts of the initial author because their mail-addresses and names all resemble the same name and a total of

1.844 commits were made by these accounts. The result set for this repository does not contain any matches for the LC.

Security Code Clone Detection

In the whole project, 6.572 CCD matches were found, 6.042 from within commits of the 8 user accounts apparently owned by Sam Judd. That is 91,94% of all results. This does not mean, that he authored the apparently vulnerable code, but that he frequently worked on files containing matches.
When two related commits contain data about the same file, with no other commit between them referencing the said file, they will be called file-consecutive commits from here on.

To gather information about vulnerabilities, the commit containing a vulnerability for a specific file for the first time in a chain of file-consecutive commits is identified. This is done by backtracking the commits via their "parentCommit" column. Through this column, an ordered chain of commits can be assembled. When two file-consecutive commits have differing match sets, that means that at least one new vulnerability has been added or that at least one has been removed. If the older of those commits contains matches for file X with vulnerabilities A and B, and the newer one contains only a match of X with A, that means that X is no longer matching with B and thus, has been fixed by the newer commit. This algorithm is run for every single CCD result - match or non-match. Using the same method, but with reverse chronology, vulnerability-introductions are detected as well. If none is found, the first commit containing the file at all is considered the vulnerability-introduction.

Two sets of introducing commits will be distinguished here. The first set contains the first occurrence of any matched vulnerability, from now on called type A commits. The other is containing all commits that introduced those vulnerabilities that have been detected as fixed in later commits, from now on called type B commits. For this, a vulnerability is considered to be the same across commits, if it is matched in an uninterrupted chain of file-consecutive commits. 77 of the 2.027 commits containing any files with relevant file endings had apparent fixes in them. Files that have no relevant file endings include files for localization, pictures, and other non-code. On average, the fixes took 301 days. Of these commits, none contained any keywords detected by the algorithm. The CWE entries that are supposedly fixed here have the ids: 611, 295, 200, 667, and 79. In comparison to the CWE-ids that have been matched, only CWE-310 has not been fixed in any case. This CWE-id is related to cryptography[72] - This information is not from the result-DB. The 77 fixes have been committed by 14 distinct users and backtracking the fixing commits to the first occurrences shows, that 7 users committed type B commits. In the set of users with type B commits, Sam Judd appears thrice, and a total of 59 such commits have been made by him. In the set of vulnerability fixing users, Sam Judd appears twice, with a total of 63 of the 77 fixing commits. All other accounts combined committed 4 vulnerabilities and contributed 14 fixes. 17 commits

have both introduced (type B) and fixed a vulnerability. The number of distinct files processed for the repository is 758 with an average of 9,21 matches per file. 388 of the files were never matched with any vulnerability. The average number of matches per file, limited to files that ever matched at all, is 18,87 matches per file. The average CVSS ratings for the matches - where applicable - are 4 for the "baseScore", 2,9 for the "impactScore", and 8,01 for the "exploitabilityScore". This measure is calculated from the 632 matches originating from the GitHub extraction tool and its counting duplicate matches among multiple commits, too. To not have one vulnerability outweigh others because of their presence in more commits, here the values can also be grouped by the combination of the input file name and the security clone example file name. This drops the match-count from 632 down to 54 distinct matches with rating values of 4.01 for the "baseScore", 2,9 for the "impactScore", and 8,01 for the "exploitabilityScore". Only two ratings are referenced by these matches, of which one occurs 53 times and the other once in the distinct matches. In all matches, the more frequent one occurs 619 times and the other 13 times. Through this, the average scores are dominated by the more frequent one.

5.2.2 Python Repository

The chosen repository is the one with the id 12. Its name is flask[73] and it has 3.798 commits. The creator of the repository is Armin Ronacher (armin.ronacher@active-4.com). It was created on the 6th of April 2010. A total of 692 users have contributed to the repository so far. 496 of those made only one commit each. As opposed to the Java excerpt, the creator of this repository does not have any obvious duplicate user accounts. Of the total 3.798 commits, he created 1.183. No LC matches have been reported for this repository.

Security Code Clone Detection

In the whole project, 106 CCD matches were found, 35 of them committed from the author's account. That is 33,02% of all results. Of the 2.093 commits containing any files with relevant file endings, 3 contained apparent fixes. Of these fixing commits, one contained a keyword match. The keyword was "SQLI" and matched with the word "SQLite", which is a false positive, as "SQLI" is supposed to mean "SQL Injection". 507 is the only CWE-id reported for this repository. The 3 fixes have been committed by 3 distinct users and backtracking the fixing commits to the first occurrences shows, that 2 users made type B commits. The fixes took 456 days on average. The repository's creator committed 1 fixing and 2 type B commits. One only other type B commit has been done by a different user and the two remaining fixes were committed by two more users. No type B commit also fixed a vulnerability. The number of distinct processed files for this repository is 118, with 116 of them having not a single match. On average, each file had 0,9 matches and if only files with any matches are considered, the average

is 53 matches per file. The average CVSS base rating of all matches is not obtainable, because all matches happen to be exclusively with code examples from StackOverflow.

5.2.3 JavaScript Repository

The chosen repository is the one with the id 33. Its name is reveal.js[73] and it has 2.391 commits. The creator of the repository is Hakim El Hattab (hakim.elhattab@gmail.com). It was created on the 7th of June 2011. A total of 301 users have contributed to the repository so far, 208 of those made only one commit each. Again, the author of the repository committed from multiple accounts. Of the total 2.391 commits, he created 1.797.

Security Code Clone Detection

In the whole project, 858 CCD matches were found. 651 of them from the author's account. That is 75,87% of all results. Of the 1.571 commits containing any files with relevant file endings, 6 contained apparent fixes. Of these commits, none contained keywords. All 6 fixes reference the same CWE and matched two files. All 6 fixes were backtracked to a single type B introduction commit and took 274 days on average to occur. The reason for this not to be just two fixes is that multiple example files for the same CWE-id are counted separately - so these 6 fixing commits fix matches with 6 different example codes for two vulnerabilities. The CWE-ids reported for this repository - not only the fixes - are all CWE-345. Of the 6 fixes that have been committed, 4 were committed by the repository's author and the only type B commit was done by him as well. Since there was only one type B commit, there was no intersection between type B commits and fixing commits. The number of distinct files processed for this repository is 46, with only 3 of them ever matching. This is an average of 18,65 vulnerability matches including all scanned files, and 286 if excluding files that never had any matches. The CVSS cannot be used for this repository's CCD matches, as it did not match any examples originating from GitHub.com.

Library Checker

From this repository, 59 distinct matches were found by the LC. None of these matches share the same CVE-key, but the number of distinct CWE-ids referenced is 22. The ratings of all found vulnerable libraries are 5,63 for the "baseScore", 8,81 for the "exploitabilityScore", and 4,59 for the "impactScore". Of the 59 matched libraries, no libraries have been detected to be fixed. All 59 matches have been introduced in a single commit.

5.2.4 Mixed language Repository

The chosen example for a mixed repository has the id 58 and contained matches for Java and JavaScript. Its name is apollo[73] and it has a commit count of 2.328. The repository was created on the 3rd of April 2016 by Frankie Wu(qmwu2000@gmail.com). A total of 93 users have contributed to the repository so far, 54 of those made only one commit each. The author appears to be using one account only and committed 2 times.

Security Code Clone Detection

In the whole project, 1.314 CCD matches were found and none was within one of the author's commits. Instead, a user called Jason Song committed the most results, with a count of 508 matches in 1.176 commits, which is 38,66% of all matches and 50,52% of all commits. Of the 964 commits containing any files with relevant file endings, 13 contained apparent fixes. Of these fixing commits, none contained keywords, and they took 285 days on average. For this repository, nine distinct CWE-ids were reported. The author contributed neither fixes nor vulnerabilities to the repository, whereas Jason Song contributed 6 fixing and 4 type B commits. There is no type B commit that is also fixing matches at the same time. The difference to the "pure" repositories discussed, is that the results of this one can be separated by language. The number of Java matches is 1.170. That is 89,04% of all matches. The remaining 144 matches were JavaScript matches with 10,96% of all matches. The total number of distinct matched files is 20 for JavaScript, 258 for Java, not counting unmatched files, as these entries do not have an associated language. That is a match-frequency of 7,2 for JavaScript files and 4,53 for Java. Regarding the CVSS ratings of the matches, the combined ratings for both languages are 4,23 for the "baseScore", 3,18 for the "impactScore", and 8,06 for "exploitabilityScore". Separated by language, the ratings for Java matches are 3.99 for the "baseScore", 2,9 for the "impactScore", and 7,98 for the "exploitabilityScore" over 815 matches. For JavaScript, the average ratings are a 10,0 for all three scores over 34 matches, which is because these 34 matches are all the same one occurring in different commits. To not have one vulnerability outweigh others because of their presence in more commits, here the values are grouped by the combination of the input file name and the security clone example file name. For JavaScript, this reduces the match-count to 1, so the rating does not change. For Java, the matches are reduced to 191 and the average rating changes to 3,98 for the "baseScore", 2,9 for "impactScore", and 7,95 for "exploitabilityScore". Only two ratings are referenced by these 191 distinct matches, of which one occurs 189 times and the other just twice, making the score of the first one dominate the average values.

Library Checker

In this repository, a single library was identified as malicious in 8 commits. The matched library is "jquery.js 1.10.2". One CVE entry was found for it: "CVE-2017-16045". The first occurrence of it within this repository was in March of 2016. This CVE is assigned to CWE-200[72], which is the code for an "Information Exposure" - this information is not from within the result-DB. In this case, no fix has been found. Looking the repository up on GitHub.com reveals, that Maven is used. Through its name's ending, "jquery.js" can easily be identified to be a JavaScript library. This means since the only source for JavaScript-related LC results is NPM, that within a single repository, two library management systems are present: Maven[35] and NPM[20].

5.2.5 Combined Excerpts

In the following tables, the results of the four excerpt repositories will be summarized, starting with general data in table 5.2.

	Java	JavaScript	Python	mixed
repository name	glide	reveal.js	flask	apollo
commits	2.457	3.798	2.391	2.328
contributors	162	301	692	93
multiple accounts of the author	yes	yes	no	no

Table 5.2: Result Excerpts' general data

Security Code Clone Detection

The CCD-results differ largely among the excerpts, with some fields not being covered at all by some. Just like in the design for the result database, this data is meant to be as broad as possible, while not expecting every value to have a particular meaning.

From our chosen repositories, a wide variety of data has emerged. The number of files, for example, appears to be very small in "reveal.js" with only 46 files, as opposed to the 758 files from "glide". That, and the fact, that "reveal.js" had the most processed commits, leads to a very high number of matches per file in said repository. In the Python excerpt, only two files matched at all, with both referencing the same CWE-id, closely followed by three matching files in the JavaScript excerpt. For Java, the rate of matches looks to be way higher, with more than 50% of the files having any match. Across the board, almost all occurring CWE-ids have also been fixed at some point.
The analysis of the CVSS-ratings of the matches is limited by the number of matches with security code examples associated with CVSS. Because of this, the CVSS analysis

	Java	JavaScript	Python	mixed	
				Java	JavaScript
repository name	glide	reveal.js	flask	apollo	
processed files	758	46	118	578	
never-matching files	388	43	116	300	
matches	6.572	858	106	1.170	144
author's contribution (matches)	6.042	651	35	0	
commits with matches	2.027	1.571	2.093	313	57
matches per file	9,21	18,65	0,9	2,27	
matches per files without never-matches	18,87	286	53	4,53	7,2
fixes	118	6	3	11	4
commits with fixes	77	6	3	9	4
author's fixing commits	63	4	1	0	0
fixing users	14	3	3	6	2
average fix time (days)	301	274	456	382	18
fixes per match	1,8%	0,7%	2,83%	0,94%	2,78%
number of matched CWE	6	2	1	7	2
number of fixed CWE	5	1	1	5	1
CVSS-containing matches	632	0	0	815	34
distinct CVSS-containing matches	54	0	0	191	1
CVSS-containing matches distinct by CVE	2	0	0	2	1

Table 5.3: Result Excerpts' CCD results

will only be done for the summary of all repositories, seeing as only half of the excerpt repositories qualify at all.

Library Checker

The LC's results are badly represented by the four excerpt repositories since the only matches were related to JavaScript. Because of this, Python and Java will not be present in table 5.4.

Of the 60 total matches, 23 distinct CWE references emerge, without a single fix detected. The three CWE-ids occurring the most frequently are 79, 310 and 330.
CWE-79 stands for XSS, which allows manipulation of web pages through user input[72].
CWE-310 is a CWE category, summarizing cryptographic issues[72].
CWE-330 stands for usage of random values that are not truly random, allowing users to predict their values in some cases[72].
Among all these matches, the impactScore never surpassed the exploitabilityScore.

	JavaScript	mixed	combined
repository name	reveal.js	apollo	
commits with matches	233	8	241
commits with fixes	0	0	0
distinct libraries matched	59	1	60
distinct CWE referenced	22	1	23
average baseScore of CVE	5,63	5,0	5,32
average exploitabilityScore of CVE	8,81	10,0	9,41
average impactScore of CVE	4,59	2,9	3,75

Table 5.4: Result Excerpts' LC results

Keyword Detection

From the four excerpt repositories, a total of 114 commits contained any keyword matches, which, of our 10.974 total commits, is only 1,04%. The intersection with commits, the CCD found matches in, contains 111 commits. Of those 111 commits, many CCD results did not have associated CWE-ids, 36 were containing the word "race" and 18 contained the word "security". Five other keywords were each only connected to CWE-related CVE in less than 10 cases each. The occurrences of "race" were connected to 7 distinct CWE-ids and of those, none matched the CWE-ids of race conditions. The occurrences of "security" were connected to 5 distinct CWE-ids, but since the keyword is in the category "Common keywords", no CWE-ids are associated with it and no information is gained. In fact, regarding keywords, for the following analysis of all repositories, the categories "Common keywords" and "Custom List" will be omitted, because they contain no CWE reference. Also, all matches of the keyword "SQLI" will be removed, if their commit messages contained the word "sqlite".

5.3 Compiled Results

General Data

In the following table, the results of the three languages will be summarized.
Here, we can already answer our first research question:
RQ1: How many applications contain security code clones?
From our 159 scanned repositories, 111 repositories contained CCD matches and 82 contained LC matches. The precise absolute and relative amounts of result-producing repositories for each language in combination with each, LC and CCD, can be seen in table 5.5.

Three repositories have been found to not contain any files with endings belonging to

	Java	JavaScript	Python	None
total repositories	159			
total commits	400.268			
repositories by most used language counted manually	51	57	49	
repositories containing relevant files	62	100	62	3
repositories containing CCD matches	52	69	23	48
repositories containing CCD matches (%)	83,37%	69%	37,1%	30,19%
commits containing relevant files for CCD	48.725	82.889	117.654	151.427
commits containing relevant files for CCD per repository	785,89	828,89	1.897,65	
commits containing CCD matches	17.874	15.221	1.969	365.393
commits containing CCD matches (%)	36,68%	18,36%	2,38%	91,29%
repositories containing LC matches	13	61	16	77
repositories containing LC matches (%)	20,97%	61%	25,81%	48,43%
commits containing LC matches	6.658	56.817	49.122	335.504
commits containing LC matches per repository	107,39	568,17	792,29	

Table 5.5: All Results' general data

one of our three languages, albeit having been selected because of their language tag. Their ids in the database are 17, 110 and 154.
The repository with the id 17 is called shadowsocks[73] and it appears, that the repository has been wiped clean for legal reasons, with the single remaining commit being commented on in Chinese.
The one with the id 110 is called funNLP[73] and almost exclusively contains ".txt"-files, filled with text that is identified as Chinese by "Google Translate"[74].
The id 154 belongs to a repository called clean-code-javascript[73] and only contains text files with content discussing JavaScript code.

Counting the repositories by language does not necessarily work, because repositories are not always restricted to a single language, hence the sum of the repositories per language, plus the repositories without any CCD matches - a total of 192 - is larger than the number of repositories processed. Even, if the language used by the "Git Crawler", to start the process for each repository, had been persisted, the same would have happened, as repositories containing files for multiple languages tend to be tagged for all these languages, too. This makes it hard to process a balanced number of repositories for each language. This is why the repositories have been counted by languages manually, using only the most used language displayed on each repository's main page. The processing of the repositories is started balanced among the languages, but that does not provide any certainty of balance for the emerging results.

Security Code Clone Detection Results

48 repositories, including the 3 that did not contain relevant files, did not produce any matches with the CCD. Of the remaining 111 repositories, the most matches were found for Java. The number of Python CCD matches is significantly lower than those for the other two languages, with a match-frequency of about 0,02 as opposed to 1,02 for Java and even 1,09 for JavaScript.
Also, it appears, that the security code examples extracted from GitHub.com have a low match-rate throughout all languages, with Python not containing a single match for those. Sadly, this diminishes the significance of the CVSS analysis, since these results are the only ones with associated CVSS ratings. For Python, this means, that the CVSS rating cannot be discussed at all. This very low rate of CVSS containing matches does not seem to be related to the portion of CVSS containing security code examples, and since the CCD has been evaluated for each language, this must be related to either the CCD approach, differences in code style among the sources, or a combination of both.

Now, to answer the research questions RQ2 and RQ3 with the CCD result data:

RQ2: Which programming languages are susceptible to which CWE?
This and the following question should be answered for both, CCD and LC, and both answers for each will be combined at the end of the section. To first answer, based on the CCD results:
From the gathered data, a top three list of matched CWE-ids has been created for each language, without any intersection of CWE-ids between the languages' top occurrences.

Java appears to be especially susceptible to information exposure due to lacking authorization systems (CWE-200)[72], lacking certificate verification (CWE-295)[72], and improper locking of resource access (CWE-667)[72].

For JavaScript the most frequent CWE-ids were uncontrolled resource consumption (CWE-400)[72], insufficient verification of data authenticity (CWE-345)[72] and the category CWE-264[72], which groups CWEs concerning lacking permission systems.

The most frequent CWE-ids for Python were race conditions (CWE-362)[72], Cross-Site Request Forgery (CSRF) (CWE-352)[72], and trojan horses (CWE-507)[72].

RQ3: Which coding language has more vulnerabilities?
Based on the frequency of matches per scanned file, the language with the most matches, and thus vulnerabilities, is JavaScript. Though, if deciding by the portion of repositories that had any vulnerability in their history, Java takes the lead, with 83,37% of all language-relevant repositories containing a CCD match.

	Java	JavaScript	Python	Combined
matches	91.480	93.590	2.323	187.393
fixes	1.289	985	70	2.344
commits with matches	17.874	15.221	1969	35.064
commits with fixes	1.021	892	70	1.983
fixes of matches (%)	1,41%	1,05%	3,01%	1,25%
processed files	90.082	86.056	106.511	282.649
distinct matching files	71.342	66.679	2.161	140.182
matches per file	1,02	1,09	0,02	0,66
matches per file excluding never-matches	1,28	1,40	1,07	1,34
never-matching files	18.740	19.377	104.350	142.467
average fix time (days)	304,35	283,05	351,71	286,99
number of matched CWE	21	10	3	29
number of fixed CWE	16	6	3	22
fixing users	291	300	40	616
security code examples	198	38	48	284
security code examples with CVSS data	50	15	12	77
security code examples with CVSS data (%)	25,25%	39,47%	25%	27,11%
CVSS-containing matches	3.424	456	0	3.880
CVSS-containing matches (%)	3,74%	0,49%	0%	2,07%
distinct CVSS-containing matches	3.375	420	0	3.795
CVSS-containing matches distinct by CVE	82	36	0	118
most frequent CWE occurrence id occurrences	200 31.804	400 44.016	362 1.241	400 44.016
second most frequent CWE occurrence id occurrences	295 22.440	345 27.368	352 649	200 31.804
third most frequent CWE occurrence id occurrences	667 20.369	264 19.109	507 433	345 27.368

Table 5.6: All Repositories' CCD results

Library Checker Results

The LC results have not been separated by language within the tool, and the CPE entries do not contain an assigned language either. The solution is, to assign languages to LC results, using the languages of source code found within the same repositories. This means, that, for example, the 49.122 LC matches that are identified as Python-related are not necessarily really Python-related, because the count can originate from repositories that contain another language next to Python. Of the 156 repositories, not counting the 3 "empty" ones, 74 did not yield LC matches. In the remaining 82 repositories, the LC results were dominated by JavaScript. Not only did the most commits contain JavaScript-related LC matches, but grouping the results by CVE-id enlarges the gap even more. JavaScript matched 85,6% of all distinct CVE-ids and 90,63% of

all distinct CWE-ids. One detail that catches the eye is, that, as opposed to the 1,25% average fixed CCD matches, 59,05% of all LC matches appear to have been fixed.
To answer the research questions RQ2, RQ3, and RQ4 with the LC result data:

RQ2: Which programming languages are susceptible to which CWE?
The LC results share a few CWE-ids among the languages. There is a chance, that this is because the matches are shared via multi-language repositories. Most frequent for Java and in second place for JavaScript is "Improper Input Validation" (CWE-20)[72]. The same constellation, but in reverse goes for XSS (CWE-79)[72] in first place for JavaScript and second for Java. In the third spot for Java and Python is CWE-264[72], which did also appear in the third place of the JavaScript CCD results. The remaining three are CWE-59[72] in the first place for Python, which is about insecure file-links, CWE-502[72] in its second place, which is about deserialization of untrusted data, and CWE-400[72] for the third place for JavaScript, which we saw on JavaScript's first place in the CCD results.

RQ3: Which coding language has more vulnerabilities?
Here, JavaScript takes the crown with its 85,6% of distinct CVE-ids matched and 58 distinct CWE-ids. This time, this result is not diminished by the limited security code example base, but instead, it might be related to how often developers use NPM as opposed to "requirements.txt", and readily packaged JAR files in Java.

RQ4: Which libraries introduce vulnerabilities into software most commonly?
Grouping the libraries by the languages of code files present in the same repository showed a satisfying result at first.

First, the appearances of CPE entries have been counted, separated by language. To then identify the three libraries with the most matches, the list of these CPE has been sorted, and the libraries have been investigated starting from the most frequent associated CPE. This revealed, that the language association did in fact not result in a sufficient distinction, because the top 2 libraries for JavaScript were Python libraries. After sorting out the wrongly sorted libraries, the top 3 libraries matched for JavaScript were "kerio_mailserver" with a count of 138.208, "cron" occurring 118.188 times, and "express" with 78.677 appearances. Analyzing the Python languages revealed that the LC results are vastly dominated by Python with a count of 553.956 for its top matching CPE "lxml". The next most frequent CPE occurrences were "requests" with 221.090 occurrences, and "virtualenv" with 175.343 occurrences.

Sadly, this investigation of each particular library revealed, that there was not a single real LC result for Java, and all results counted as Java-related were originating from mixed repositories. This can be explained with the tactic used for the Java LC, as it is targeting ".jar"-files, which only rarely occurs in repositories at all. After manually sorting all 159 repositories by their most used languages and counting the ".jar"-files used in all

Java repositories, a possible explanation was found: The number of repositories ever containing ".jar"-files is 25. Of those jar files, the two most frequent files were wrappers for Gradle[36] and Maven[35]. After sorting these out, only 8 repositories contained ".jar" files. Looking at these remaining repositories' ".jar" files, showed mainly libraries that might have been LC matches themselves, but apparently did not serve as sources of library lists.

One information drawn from this - the libraries associated with Java were almost exclusively from JavaScript, so it seems that a mixed repository between Java and Python is much rarer than one between Java and JavaScript.

	Java	JavaScript	Python	Combined
matches				278.103
commits with matches	6.658	56.817	49.122	64.764
commits with fixes	179	3.972	389	4.155
distinct matches	436	2.266	257	2.647
distinct fixes	204	1.422	183	1.563
fixes of matches (%)	46,79%	62,75%	71,21%	59,05%
distinct matching CWE-ids	43	58	46	68
distinct fixed CWE-ids	32	58	42	64
distinct matches per CVE	10,14	47,21	5,59	38,93
average fix time (days)	205,83	220,12	206,83	206,83
most frequent CWE occurrence id occurrences	20 6.216	79 48.767	59 27.596	20 56.273
second most frequent CWE occurrence id occurrences	79 5.290	20 47.001	502 19.364	79 49.827
third most frequent CWE occurrence id occurrences	264 3.328	400 28.347	264 18.234	400 31.841

Table 5.7: All Repositories' LC results

Ratings

Here, Java will be excluded from the following LC table 5.8, because there were no real results for Java. The three scores, impactScore, exploitabilityScore, and baseScore will be abbreviated in the two following tables 5.9 and 5.8.
None of the LC-results for Java were true positives, with the only alternative for the source of the library matches being multi-language repositories. This implies, that all libraries matched for Java appear in the result sets of the other two languages. The "Combined"-row is calculated based on an intersection between all result sets, meaning that duplicate matches from multi-language repositories will be excluded from its calculation.

Table 5.8 shows, that across all matches, no difference larger than 0,05 between languages exists for all scores. The only real difference is in the "Unfixed Matches"' scores, with a maximum distance of 0,4. It appears, that the choice of language poses no real influence on the outcome of library ratings.

	All matches			Unfixed Matches		
	eS	iS	bS	eS	iS	bS
JavaScript	8,57	4,48	5,43	8,46	4,47	5,38
Python	8,52	4,47	5,41	8,2	4,87	5,54
Combined	8,57	4,48	5,43	8,46	4,47	5,38

Table 5.8: LC CVSS scores for all repositories

The following table 5.9 shows the averaged CVSS ratings of the few CCD-matches with security code examples originating from GitHub.com. With no particular intent and a maximum difference of 0,51 between "All Matches"'s and "Distinct CVE"'s exploitability scores, the distinction becomes insignificant. This time, like in the LC-based CVSS analysis, one language is missing applicable results. Instead of Java, Python is lacking, because the CCD results of Python did not contain matches with security code examples originating from GitHub.com, as explained in section 2.7.3.
Here, as opposed to the LC results, the rating shows big differences between the two languages, with Java having a lower score in every single field. The number of CVSS-bearing results, that have been used in the calculation of these values can be seen in table 5.6. The difference in the scores could be explained through the differing amounts of results used for their calculation. Otherwise, this implies, that vulnerabilities related to JavaScript are not only more accessible but also more dangerous. These CCD-based CVSS results have to be taken with a grain of salt, as the CVSS scores come from CVE references that are designed to reference software and particular releases - not pieces of code. So what has been done here, is perverting the CVSS rating into a possibly valid variation.

	All Matches			Distinct Files			Distinct CVE		
	eS	iS	bS	eS	iS	bS	eS	iS	bS
Java	7,87	2,92	3,96	7,84	2,92	3,94	7,56	2,97	3,94
JavaScript	9,15	4,56	5,76	9,15	4,61	5,8	8,64	4,76	5,8
Combined	8,3	3,45	4,55	8,29	3,46	4,55	7,94	3,58	4,55

Table 5.9: CCD CVSS scores for all repositories

Keyword Detection

The keywords detected from commit messages have been grouped by the CWE-ids of the CCD matches within the same commits. The keywords were initially assigned with a selection of CWE-ids, as stated in the paper the keywords originate from[51], and an intersection of those ids and the matches within the same commits has been anticipated. The largest intersections are with the keyword "race", with 3 occurrences in commits that did contain matches for CWE-362[72], which is referencing race conditions, and with "DOS", with 7 occurrences connected to CWE-400[72]. But since CWE-362[72] has been matched 1.241 times by the CCD, and CWE-400 even 44.016 times, these occurrences are likely coincidences and definitely not fit to detect matching code.

Considering the LC results, three keywords emerged with any correlation to their CWE-ids. The keyword "permission" is correctly matching 182 times in 486 commits, "DOS" matches 33 in 209 occurrences, and "XSS" matches 7 out of 59. All three of these keywords are assigned to groups of CWE, so if one was to attempt finding vulnerabilities through keywords, only such CWE-groups appear to be detectable with a reasonable successrate. The keywords are related to 14 CWE-ids for "permission", 10 for "DOS", and 9 for "XSS", matching the trend of the keyword match-rate - more CWE-ids lead to a higher match-rate. Two more keywords were referencing the same group as the keyword "permission" - "gain access" and "unauthenticated" -, though neither produced any matches.

RQ5: Is the size of a repository related to its susceptibility to vulnerabilities?
To answer this, two graphs have been created. In them, one dot has been drawn for each repository, grouped by languages. Additionally, linear regression has been applied to the dots, and a trend of maximum values for combined results has been added to both graphs. Again, Java is not included in the LC graph due to lacking results. The graph based on the LC results can be seen in figure 5.1, and it shows that most repositories form a dense area in the low file count across all library frequencies. Only a few outliers can be seen further to the right and it looks like the frequency of matches is lower for repositories with more files, though, for such repositories, there are too few examples to be representative. The "combined" regression line descends, even though both other lines do not. This is only possible because of the vague language association, through which a significant portion of libraries occurred in multiple languages and distorted the lines. The "combined" data is not deduced from the others but calculated independently to rule out these faulty associations, making it the most valid line in the graph. The regression lines do not resemble the dots for each set. Because of this, they are likely invalid as an actual trend. To investigate this further, more repositories with high file counts would need to be scanned.

The graph for the CCD results regarding project size can be seen in figure 5.2. There, both the trend lines for Python and Java seem more fitting to the dots, than the one for JavaScript. Just like the graph for the LC results, here the most repositories are shown to have less than 2.000 files with a quite even distribution over the match-frequencies.

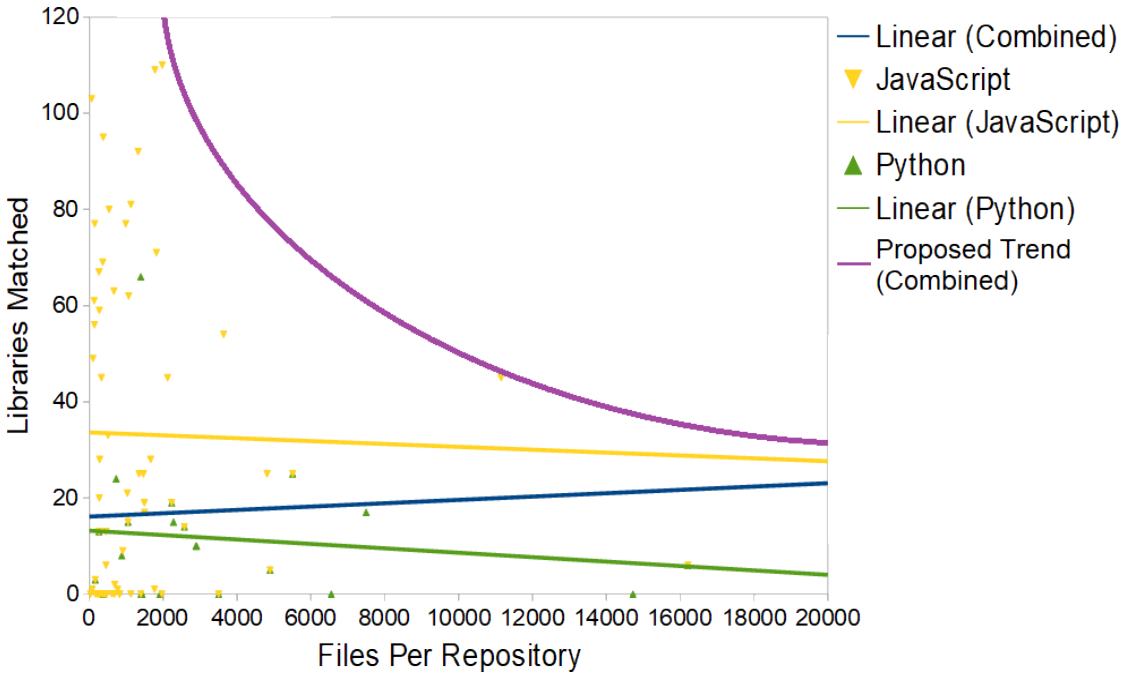

Figure 5.1: Relation of repository size to vulnerability occurrences in LC

But since the number of repositories containing any match is larger for the CCD, and since Java has results as well, more repositories with a higher file count can be seen. As always, the match-frequency for Python code is in the low regions. The linear regressions do appear to be less authentic as a prediction of vulnerability, than the purple "Proposed Trend" line. More files allow for smoother average values. Because of this, a declining average vulnerability count, converging towards an unknown value, does fit the expectations. This unknown value could be determined in a larger study, and would likely show different results for each language.

CCD and LC answers combined

To finish the result discussion, RQ2 and RQ3 will be discussed again, combining the answers from the LC- and CCD-sections.

RQ2: Which programming languages are susceptible to which CWE?
In all five top 3-lists, excluding the Java LC results and the Combined lists, 13 distinct CWE-ids are represented. Each of them has been discussed in the respective section, but the two CWE-ids that did occur in both, LC and CCD, were 264[72] and 400[72].

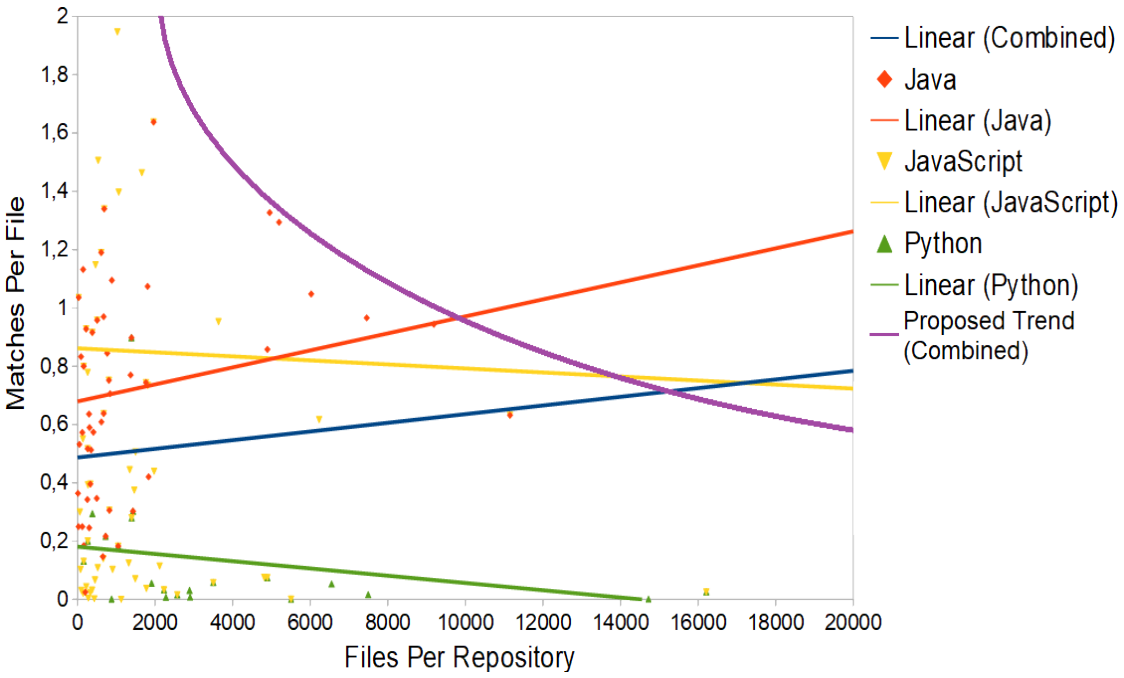

Figure 5.2: Relation of repository size to vulnerability occurrences in CCD

CWE-264[72] is the category for lacking permission systems. It appears in the top list of both JavaScript in the CCD and Python in the LC. CWE-400[72] is about uncontrolled system-resource consumption and appears in JavaScript's top 3-list both times.

RQ3: Which coding language has more vulnerabilities?
Based on both, the CCD and the LC result set, JavaScript emerges as the most vulnerable of the three tested programming languages. In the CCD results, JavaScript produced 6,86% more matches per file than Java, and 54,5 times the matches per file compared to Python. In the LC results, it has matched the most CWE-ids among the three languages with a count of 58 distinct CWE-ids and it has by far the highest percentage of repositories matching any library. Also, fixing LC matches takes the longest for JavaScript with 220,12 days on average, taking 6-7% longer than both other languages' fixes. It cannot be ruled out, that Java would have been the most vulnerable language in the LC result set if a more fitting LC tactic had been used for it.

5.4 Threats to Validity

Since the results are separable entirely by LC, CCD, and keyword detection, threats do their validity will be discussed separately.

5.4.1 Code Clone Detection

The utilized token-based CCD uses a statistical measure to identify code clones. This means, that, without manual checking, not a single match - not even type 1 - is certainly a match. Also, the security code clone examples are not verified, meaning that the found matches - if they are true positives - are matches to the examples, but not necessarily to valid vulnerable code.
Another issue with the security code examples is, that they are handpicked and not comprehensive. The results are thus based upon security code examples with a very limited number of CWE-ids referenced. This way, it is not even ensured, that among the examples of our three languages, the selection of CWE-ids intersects at all. This means, that if, for example, CWE-20[72] appears as the most common threat in Java code, this might be the case in both other languages, without this study pointing it out, because there are no security code examples connecting these languages to CWE-20[72]. Through the means of security code example creation utilized in this thesis, another problem emerged, with the examples having differing metadata with the code examples from the "StackOverflow Extractor" being unfit for CVSS association.

The most frequent matches for Python had trojan horses among their top three. As this is unlikely to be true, an alternative explanation would be, that the trojan horse code example is a large code example file, containing 6 compound statements, all pointing towards the same file upon matching. This effect is amplified by the low count of CCD results for Python.

5.4.2 Library Checker

In the LC result data, the file that contained the processed library names has not been persisted. This leads to three weaknesses in the data analysis that should be fixed in the future:

Firstly, the backtracking used to identify fixed library vulnerabilities will not "know" about commits that did include changes to the libraries but matched none. For example, if five libraries were within such a file, three of them producing matches were changed in a commit, removing one of those libraries entirely, the algorithm would correctly identify the fix. If all three were removed instead, the commit would not produce an LC match at all, which the algorithm would interpret as if the commit did not contain a relevant file at all, and thus, not result in correct backtracking.

Secondly, if multiple files were used for library management, a similar problem would emerge. One file, containing 6 matches, and another, containing 8 matches. In one commit, both are changed and the LC reports 14 matches. In the next commit, only the first of the files is altered, removing one of the matches. The LC would report five matches. The backtracking algorithm would deduce from that, that 9 libraries have been fixed, and reintroduced the next time changes are committed for the second file. Another problem is, that the fuzzy queries of the LC include versions that were not actually used.

Consider this example: A single library is mentioned in one commit, including a version identifier, producing a match with the NVD. This match is not ensured to be a match for that particular version. Should the specified version not occur in the NVD, the query would instead return all CPE with the same library name, ignoring potentially mismatching version identifiers. Now, in another commit, the version identifier of the library is changed. The backtracking algorithm matches LC reports using the CVE URIs. If the first occurrence matched CVE URIs X, Y, and Z, the particular version has not been found. In the second occurrence, it matched X, Y, and Z, too, because its version, albeit being a different one, still did not match any CPE. In its third occurrence, the version changes again. This time the version matches a particular CPE. The only match produced is X. Through the algorithm, Y and Z are identified to have been fixed here, though this is the first time, that a match had a version-including match. Fixing this could prove quite difficult, as the format in which versions are identified is not unified. The version identifiers can be anything from numbers to numbers with multiple dots as separators, text, Chinese letters, or anything else displayable using bits and bytes. In current culture, it would not be surprising to see people choose Unicode characters as version identifiers for their software just to mess with other people's algorithms.

The third problem emerging is, in the LC, no languages are assigned to the results, even though the LC extraction is per-language, so at that point, the matched language could have been added to the result data.

The LC result set did not contain any results for Java, because of the way the LC module for Java works. It was a faulty concept from the start because it is based on ".jar"-files that rarely occur in repositories at all.

5.4.3 Keyword Detection

The main problem with the keyword detection is the low number of keywords. The set that was looked for in commit messages, contained only 26 keywords, of which only half were assigned with CWE-ids. Instead of looking for a fixed set of keywords in the commit messages, the commit messages could be parsed into a much larger term frequency map. This map could then be combined with the set of occurred vulnerabilities within the same commit. Counting the resulting connections between words and vulnerabilities could produce a significant selection of keywords, of which each would have

an F1 score. Many words would match each and every vulnerability, because they are common words and appear in next to every commit message, while some might show to be very precise.

6 Related Work

Previous Work

This thesis is based on work done preliminarily mostly as theses of students.

Evers[32] and Müller[31] created the tools for the security code example extraction from GitHub.com, StackOverflow.com and more. These tools have been used in another tool developed in the Software Engineering Group at Leibniz University Hannover[30], without an associated paper or thesis, to create CCD example code. Wagner[22] and Viertel et al.[21] introduced the first iteration of the LC used in this study. In that version, the only supported language is Java and there is a hook to take secondary vulnerability databases other than NVD into consideration, which has been removed in our case. Viertel et al.[76] provided the base for our CCD-module. It uses the same reference repository, the same CCD and the same tools to gather security code examples for said reference repository. Rosenthal[75] compared multiple neural network-based approaches to CCD and offers his own attempt as well. This might prove to be an excellent option for a new CCD variant for the software, as is described in chapter 8.

Similar Goals

Sönnerup and Hell[77] applied the fifth of our discussed methods of CCD, namely the metric-based CCD, to find probable security examples. This, just like the token-based CCD works on a vague base of statistical emergences and can be useful to provide direction for manual processing, but not for large-scale automation. Jimenez et al.[78] present a tool called VulData7 that collects various information about a fixed set of vulnerabilities. It is intended to gather as much information around vulnerabilities as possible, to be used in studies like this one. The information gathered in the tool is just about the same as the information, the GitHub exporter tool used in this thesis provides after it has been combined with NVD-data, which is also done in this thesis. It might be a good choice for the assembling of data sources like our reference repository. Gkortzis et al.[79] provide a dataset based on metrics derived from looking up software reported to the NVD, enriched with associated development techniques, testing, and continuous integration. Shin et al.[6] evaluate the validity of metric-based CCD as indicators for vulnerabilities. Bosu et al.[51] analyzed vulnerable code occurrences to find factors fit to detect them. The keyword set used in our thesis has been taken from this paper.

The "Vulncode-DB"[48] while not a scientific paper, is a novel resource that is currently in its alpha phase, trying to fill the gap for reliable code examples for vulnerabilities. If

that project becomes what it aims to be, a lot of the here underlying software would be made obsolete, like the reference repository utilized in this study, as well as all the tools created to gather examples from GitHub, StackOverflow, etc. would either be used to feed that website's dataset, or become deprecated entirely. The "Exploit Database"[49] is another example of such a resource, fit to replace the reference repository entirely. It is older than the "Vulncode-DB" and both are closely linked with CVE references. "GitHub advisories"[45] is a system within GitHub.com, used to first discuss vulnerabilities within a repository discretely, which is made publicly available after being handled. These discussions are also often linked with CVE-ids. Kaur and Sharma[7] discuss the relation of clone types to CCD-techniques and define capability limits of CCD-techniques regarding the clone type hierarchy as in table 6.1.

Clone Type	CCD-Technique
1	Text Based Method
2	Token Based Method
3	AST Based Method
4	PDG Based Method
Others	Metric Based Method Hybrid Method

Table 6.1: CCD-methods assigned to Code Clone Types

This, of course, is a rough abstraction, as this only shows the expected capability of each method, ignoring the kind of data each method is able to produce. Henderson and Podgurski[1] propose an example for a CCD, based on PDG and thus, possibly a valid type 4 code clone detector. Knorr et al.[70] offer a tool to compare a dataset to the NVD, to report vulnerabilities within a system. This tool is trying to find weakness-reports for software installed on the hosting system by looking automatically gathered software identifications up in the NVD. Here two interesting claims are made:
One is, that of the CWE-database, only 29 ids are actually used, and the other, that CVE entries are often reported just for the most recent version of a CPE at the time of the report, while often the vulnerabilities are also occurring in previous versions. Also, a trend is noted, where vulnerability handling is shifting into commercial bounds, as opposed to open-source alternatives.

7 Conclusion

In this thesis, a study was conducted to answer questions about vulnerabilities occurring in code written in the three languages Java, JavaScript and Python. For this cause, a tool was developed, based on a plugin for JIRA, that was used to scan the top 159 repositories on GitHub.com for vulnerable code, vulnerable libraries, and keywords related to those. For that, the developed tool contained three modules, namely security code clone detector, library checker, and keyword detector, that each created a separate dataset. The Security Code Clone Detector[19] (CCD) uses security code examples, that are associated with vulnerability reports in the NVD, to find vulnerabilities in software, by looking for similarities between the examples and the code in the software in question. The Library Checker[21] (LC) extracts library identifications from projects to then look them up in the NVD. The keyword detector searches for occurrences of specific keywords in commit-messages, to later draw connections between those keywords and vulnerabilities within the same context. The three resulting datasets were discussed separately and occasionally combined for additional insights. Of the five research questions, all have been answered to varying extent in chapter 5.3, and additional insights have been deduced from the abundance of data produced by the tool. The result data turned out to be very comprehensive, and many more questions could have been answered through this data.

Furthermore, the developed tool poses a foundation for several improvements in multiple modules like CCD and LC, of which some are proposed in the following Future Work chapter.

8 Future Work

The software that was created for this thesis is far from perfect, but it is designed for improvement.

Performance

In our tool's current state, lots of data is written to databases or files, just to be read again. For example, the NVD dump in its full state is just 300MB in size, which would pose no threat to the capacity of a JVM. It should still be saved as a database so that a run would not require a full NVD download each time. After that, though, it should be kept within memory.

Lots of data is currently buffered within memory, to preserve the identity of rows. This is done to reduce redundancy in the result-database and ensure that datasets that are linking, for example, the same CVE do not just link two identical CVE instances, but the same one. These buffers could become significantly smaller, not only because they would not have to contain the NVD entries referenced by the results, but also, because they are not too optimized considering what to buffer, which could lead to a smaller demand of memory in total.

Another unnecessary use of file storage is, that the SourcererCC and tokenizers are storing tokens and metadata in files, just to read them again - usually within a second. This is done for every single commit, resulting in a vast number of file-reads and -writes and large optimization potential.

ANother Tool for Language Recognition

Using ANTLR for the preprocessing of code examples has already been implemented. Though currently, it is only used for Python. A dynamic system can be used, to pull all official ANTLR grammars available, since ANTLR has an official repository, containing them. All those grammars, of which there are hundreds as of now, can be dynamically used to generate ASTs for most existing programming languages - possibly even natural language. In the directory of each language, there is another directory, called examples, containing files used to evaluate the grammar. These files have file endings belonging to the respective language, allowing the ending-based file-filtering to be implemented dynamically, too. One thing that might prove difficult but very beneficial, would having

all those parsers use one common AST format so that the CCD would be completely language agnostic.

Code Clone Detection

The SourcererCC was not a decent choice for a security study, where no manual post-processing of the results is intended, as a token-based CCD produces very disputable results by design. For future work, it should be replaced entirely.

For a replacement, three options come to mind:
A machine learning-based method, the AST-based method, and the PDG-based method. Of those, the AST-based method is the one that is the easiest to implement to one's liking, for example, by enriching the ASTs of security code examples by attaching flags to their nodes. Here are two examples of such flags.

flag	meaning
'mandatory'	This node and its subtree need to be present for a match
'fix'	This node and its subtree being present means that the vulnerability has been taken care of

This would be hard to do with code that is not preprocessed into AST already, so the reference repository could be supplemented with an AST based branch containing pre-compiled AST and their respective code for transparency. Then, for different types of clone, different algorithms of clone detection would be necessary.

For type 1, the tree of the input would have to be checked for all nodes matching the type of the example's head node, then for each match, the entire subtree and select values of the nodes would have to match.

For type 2, the same algorithm can be used, but the selection of values is smaller, excluding variable/function names from the fields that have to match.

Finally, for type 3, the same algorithm as for type 2 can be used, with additional usage of the aforementioned flags. This way, the algorithm could identify subtrees that contain fixes and thus, not return matches for them. As stated in the "Basics" chapter, an AST-based CCD is incapable of detecting type 4 clones. Additionally, the question of the CCD block granularity would become obsolete, as the tree would represent the entire file, and the granularity would be defined by the type of the head node of the code example.

If like proposed in the ANTLR section above, one was able to unify all language's ASTs into a common format, another possible effect would be, that the security code examples become language independent.

The machine learning-based method is way less configurable and transparent, since it is common knowledge in machine learning, that for example neural networks quickly

escape the understanding of their creators, once trained. The benefits and downsides of that method have been described in this thesis by Rosenthal[75].

Then there is the PDG based method, that is said to be able to detect type 4 clones[7], which is its biggest advantage, though it comes with a cost of, again, transparency, simplicity, and configurability.

Example Code Base

The example code for the CCD is based on community knowledge, but from sources that are not designed as sources for vulnerable code. The users do not use the websites with the intention of displaying code flaws in their pure form.

These sources are then manually parsed by a student in cooperation with his supervisor, with none of both being a certified security expert. Most of these examples have to be reduced to their essence and fit into a format fitting the language, which can incidentally make such an example worthless. The solution presents itself through two websites - and possibly more - mentioned in the related work chapter, that are focussed on developing examples for vulnerable code and all metadata already provided, linking the code to the NVD. One could comfortably pull all examples for a select language's code vulnerabilities from such a data source, preprocess them into their own reference repository, and run an automated update from time to time. This would remove a lot of manual processing and uncertainty from the data, as long as the sources are expected to be reliable - which has to be confirmed first.

Of course, another alternative would be designing such a public resource with all the specifics that are beneficial to the cause, factually creating a database akin to the NVD.

Library Checker

While the number of languages for the CCD can be extended dynamically using ANTLR, the current concept of the LC can not. There is a different concept though, that fits the dynamic ANTLR-based CCD perfectly. The ASTs created by the ANTLR parsers will contain nodes representing "import"-statements from within the code. This might not be true for every single language, but for many languages, like Java, Python, and C, it is true. So instead of manually implementing extractors for Maven, NPM, etc., one could use those nodes for the LC, practically leveling the LC validity across all languages with such "import"-nodes.

The only downsides to this are, that these "import"-statements usually do not contain version numbers, and tend to reference files within the current project instead of libraries, which might lead to false positives, if not filtered properly. But extending the range from three to hundreds of languages might be worth the partial loss of precision - partial, because NPM and the like do not always provide version identifiers, either.

Acronyms

CCD Security Code Clone Detector[19]

NPM Node Package Manager[20]

LC Library Checker[21]

CVE Common Vulnerabilities and Exposures[15]

CPE Common Platform Enumeration[14]

CWE Common Weakness Enumeration[16]

CVSS Common Vulnerability Scoring System[17]

CVSSV2 CVSS version 2[17]

CVSSV3 CVSS version 3[18]

ANTLR ANother Tool for Language Recognition[10]

LOC Line(s) of Code

AST Abstract Syntax Tree[23]

NVD National Vulnerability Database

PDG Program Dependence Graph[1]

NIST National Institute of Standards and Technology[26]

IDE Integrated Developer Environment[27]

API Application Programming Interface[28]

JPA Java Persistence API[29]

DB Database

SE Software Engineering

CSRF Cross-Site Request Forgery

XSS Cross-Site Scripting

Bibliography

[1] T. Henderson, A. Podgurski, *"Sampling Code Clones from Program Dependence Graphs with GRAPLE", 2016*, https://hackthology.com/pdfs/swan-2016.pdf, Accessed on 09.02.2020

[2] *Source of the utilized Python ANTLR grammar*, https://github.com/antlr/grammars-v4/tree/master/python/python3, Accessed on 09.02.2020

[3] *ANTLR target languages*, https://github.com/antlr/antlr4/blob/master/doc/targets.md, Accessed on 09.02.2020

[4] *ISO/IEC 27005:2018*, https://www.iso.org/standard/75281.html, Accessed on 09.02.2020

[5] *Internet Security Glossary, Version 2*, https://tools.ietf.org/html/rfc4949, Accessed on 09.02.2020

[6] Y. Shin, A. Meneely, L. Williams, J. Osborne, *"Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities", 2011*

[7] G. Kaur, S. Sharma, *"The Survey of the Code Clone Detection Techniques and Process with Types (I, II, III and IV)", 2007*, https://pdfs.semanticscholar.org/f560/0f495f863fd9f62ed29873d509939cd09ca0.pdf, Accessed on 09.02.2020

[8] H. Sajnani, V. Saini, J. Svajlenko, C. Roy, C. Lopes, *"SourcererCC: Scaling Code Clone Detection to Big Code", 2015*, https://arxiv.org/pdf/1512.06448.pdf, Accessed on 09.02.2020

[9] *Repository of ANTLR grammars*, https://github.com/antlr/grammars-v4/tree/master/python, Accessed on 09.02.2020

[10] *Homepage of ANTLR*, https://www.antlr.org/, Accessed on 09.02.2020

[11] *Documentation of utilized GitHub.com API*, https://developer.github.com/v3/, Accessed on 09.02.2020

[12] *Homepage of GraphQL*, https://graphql.org/, Accessed on 09.02.2020

[13] *Documentation of novel GitHub.com API*, https://developer.github.com/v4/, Accessed on 09.02.2020

[14] *CPE specification*, https://cpe.mitre.org/specification/, Accessed on 09.02.2020

[15] *CVE specification*, https://cve.mitre.org/cve/identifiers/index.html, Accessed on 09.02.2020

[16] *CWE specification*, https://cwe.mitre.org/about/index.html, Accessed on 09.02.2020

[17] *CVSSv2 documentation*, https://www.first.org/cvss/v2/guide, Accessed on 09.02.2020

[18] *CVSSv3 documentation*, https://www.first.org/cvss/specification-document, Accessed on 09.02.2020

[19] P. Gautam, H. Saini, *"Various Code Clone Detection Techniques and Tools: A Comprehensive Survey", 2016*, https://www.researchgate.net/publication/312080887_Various_Code_Clone_Detection_Techniques_and_Tools_A_Comprehensive_Survey, Accessed on 09.02.2020

[20] *Homepage of NPM*, https://www.npmjs.com/, Accessed on 09.02.2020

[21] F. Viertel, F. Kortum, L. Wagner, K. Schneider, *"Are Third-Party Libraries Secure? A Software Library Checker for Java", 2019* https://www.springerprofessional.de/are-third-party-libraries-secure-a-software-library-checker-for-/16418758, Accessed on 09.02.2020

[22] L. Wagner, *"Konzept und Entwicklung eines Schwachstellenprüfers für Java-Bibliotheken", 2017*

[23] *The wikipedia page of ASTs*, https://en.wikipedia.org/wiki/Abstract_syntax_tree, Accessed on 09.02.2020

[24] *The example figure for an AST*, https://upload.wikimedia.org/wikipedia/commons/thumb/c/c7/Abstract_syntax_tree_for_Euclidean_algorithm.svg/400px-Abstract_syntax_tree_for_Euclidean_algorithm.svg.png, Accessed on 10.02.2020

[25] *The NVD homepage*, https://nvd.nist.gov/, Accessed on 09.02.2020

[26] *The NIST homepage*, https://www.nist.gov/, Accessed on 09.02.2020

[27] *The wikipedia page of IDEs*, https://en.wikipedia.org/wiki/

Integrated_development_environment, Accessed on 09.02.2020

[28] *The wikipedia page of APIs*, https://en.wikipedia.org/wiki/Application_programming_interface, Accessed on 09.02.2020

[29] *The main JPA resource*, https://jcp.org/en/jsr/detail?id=338, Accessed on 09.02.2020

[30] *The homepage of the Software Engineering Institute of Leibniz University, Hannover*, http://www.se.uni-hannover.de/pages/de:startseite, Accessed on 09.02.2020

[31] C. Müller, *"Security Code Exporter für GitHub"*, *Bachelor's thesis, Leibniz Universität Hannover, Fachgebiet Software Engineering,[30] 2018.*

[32] Y. Evers, *"Suche von sicherheitsrelevantem Code auf Stack Overflow"*, *Bachelor's thesis, Leibniz Universität Hannover, Fachgebiet Software Engineering,[30] 2018.*

[33] *Github homepage*, https://github.com/about, Accessed on 10.02.2020

[34] *Git homepage*, https://git-scm.com/, Accessed on 10.02.2020

[35] *Maven homepage*, https://maven.apache.org/, Accessed on 10.02.2020

[36] *Gradle homepage*, https://gradle.org/, Accessed on 10.02.2020

[37] *General Information about NVD*, https://nvd.nist.gov/general, Accessed on 10.02.2020

[38] *NIST Computer Security Division homepage*, https://www.nist.gov/itl/csd, Accessed on 10.02.2020

[39] *NIST Information Technology Lab information page*, https://www.nist.gov/itl/about-itl, Accessed on 10.02.2020

[40] *The U.S. Department of Homeland Security's National Cyber Security Division*, https://www.cisa.gov/cybersecurity-division, Accessed on 10.02.2020

[41] *Mitre Corporation homepage*, https://www.mitre.org/about/corporate-overview, Accessed on 10.02.2020

[42] *XML retirement update of the NVD*, https://nvd.nist.gov/General/News/XML-Vulnerability-Feed-Retirement, Accessed on 10.02.2020

[43] *Documentation on Stars on GitHub.com*, https://help.github.com/en/github/getting-started-with-github/saving-repositories-with-stars, Accessed on 10.02.2020

[44] *NIST's definition of vulnerabilities*, https://nvd.nist.gov/vuln, Accessed on 10.02.2020

[45] *GitHub.com advisories*, https://github.com/advisories, Accessed on 14.02.2020

[46] *StackOverflow homepage description*, https://stackoverflow.com/company, Accessed on 10.02.2020

[47] *StackExchange's Security focussed site*, https://security.stackexchange.com/tour, Accessed on 10.02.2020

[48] *Vulncode-DB*, https://www.vulncode-db.com/, Accessed on 14.02.2020

[49] *Exploit Database*, https://www.exploit-db.com/, Accessed on 14.02.2020

[50] *StackExchange's code reviewing site*, https://codereview.stackexchange.com/tour, Accessed on 10.02.2020

[51] A. Bosu, J. Carver, M. Hafiz, P. Hilley, D. Janni, *"Identifying the characteristics of vulnerable code changes: an empirical study", 2014*, https://www.researchgate.net/publication/286077750_Identifying_the_characteristics_of_vulnerable_code_changes_an_empirical_study, Accessed on 10.02.2020

[52] *Documentation for GitHub.com's rate-limit system*, https://developer.github.com/v3/rate_limit/, Accessed on 10.02.2020

[53] *Documentation of pip about requirements.txt*, https://pip.readthedocs.io/en/1.1/requirements.html, Accessed on 10.02.2020

[54] *ANTLR plugin for Maven*, https://www.antlr.org/api/maven-plugin/latest/, Accessed on 10.02.2020

[55] *Homepage of hibernate*, https://hibernate.org/, Accessed on 10.02.2020

[56] *Homepage of JIRA*, https://www.atlassian.com/de/software/jira, Accessed on 10.02.2020

[57] *Hibernate Figure source*, https://dzone.com/articles/what-is-the-difference-between-hibernate-and-sprin-1, Accessed on 10.02.2020

[58] *The SQLite homepage*, https://www.sqlite.org/index.html, Accessed on 10.02.2020

[59] *The JGit homepage*, https://www.eclipse.org/jgit/, Accessed

on 10.02.2020

[60] *The eclipse homepage*, https://www.eclipse.org/, Accessed on 10.02.2020

[61] *The homepage of Project Lombok*, https://projectlombok.org/, Accessed on 10.02.2020

[62] *The Maven[35] source repository*, https://mvnrepository.com, Accessed on 10.02.2020

[63] *The homepage of Spring*, https://spring.io/, Accessed on 10.02.2020

[64] *Java EE presentation*, https://www.oracle.com/java/technologies/java-ee-glance.html, Accessed on 10.02.2020

[65] *Spring Boot documentation*, https://docs.spring.io/spring-boot/docs/current/reference/html/, Accessed on 10.02.2020

[66] *Spring Data documentation*, https://spring.io/projects/spring-data, Accessed on 10.02.2020

[67] *The homepage of EclipseLink*, https://www.eclipse.org/eclipselink/, Accessed on 10.02.2020

[68] *The homepage of MySQL*, https://www.mysql.com/de/, Accessed on 10.02.2020

[69] *Manual of the top-command*, https://wiki.ubuntuusers.de/top/, Accessed on 10.02.2020

[70] K. Knorr, M. Scherf, M. Ifland, *"Automatisierte Erkennung von Sicherheitslücken mittels CVE, CPE und NVD", 2016*

[71] V. Piantadosi, S. Scalabrino, R. Oliveto, *"Fixing of Security Vulnerabilities in Open Source Projects: A Case Study of Apache HTTP Server and Apache Tomcat", 2019*

[72] *CWE dictionary pages for several ids*, https://cwe.mitre.org/data/definitions/20.html, https://cwe.mitre.org/data/definitions/59.html, https://cwe.mitre.org/data/definitions/79.html, https://cwe.mitre.org/data/definitions/89.html, https://cwe.mitre.org/data/definitions/200.html, https://cwe.mitre.org/data/definitions/264.html, https://cwe.mitre.org/data/definitions/295.html, https://cwe.mitre.org/data/definitions/310.html, https://cwe.mitre.org/data/definitions/330.html, https://cwe.mitre.org/data/definitions/362.html, https://cwe.mitre.org/data/definitions/400.html, Accessed on 11.02.2020

[73] *The four repositories discussed in section 5.2.*, https://github.com/bumptech/glide, https://github.com/hakimel/reveal.js, https://github.com/pallets/flask, https://github.com/ctripcorp/apollo, https://github.com/shadowsocks/shadowsocks, https://github.com/fighting41love/funNLP, https://github.com/ryanmcdermott/clean-code-javascript, Accessed on 10.02.2020

[74] *Google Translate*, https://translate.google.com/?hl=de, Accessed on 13.02.2020

[75] L. Rosenthal, *"A Neural Network for Code Vulnerability Analysis", 2018*

[76] F. Viertel et al., *"Detecting Security Vulnerabilities using Clone Detection and Community Knowledge", 2019*

[77] J. Sönnerup, M. Hell, *"Evaluating Security of Software Through Vulnerability Metrics", 2018*

[78] M. Jimenez, Y. Traon, M. Papadakis, *"Enabling the Continuous Analysis of Security Vulnerabilities with VulData7", 2018*

[79] A. Gkortzis, D. Mitropoulos, D. Spinellis, *"VulinOSS: A Dataset of Security Vulnerabilities in Open-source Systems", 2018*

Declaration of Independence

I hereby certify that I have written the present bachelor thesis independently and without outside help and that I have not used any sources and aids other than those specified in the work. The work has not yet been submitted to any other examination office in the same or similar form.

Hannover, February 17, 2020

_____

Simon van Schwartzenberg