Detecting Security Vulnerabilities using Clone Detection and Community Knowledge

Fabien Patrick Viertel[1], Wasja Brunotte[1], Daniel Strüber[2], Kurt Schneider[1]
[1] Software Engineering Group, Leibniz University Hannover, Hannover, Germany
[2] Software Engineering Division, Chalmers | University of Gothenburg, Gothenburg, Sweden
{fabien.viertel, wasja.brunotte, kurt.schneider}@inf.uni-hannover.de, danstru@chalmers.se

Abstract— **Faced with the severe financial and reputation implications associated with data breaches, enterprises now recognize security as a top concern for software analysis tools. While software engineers are typically not equipped with the required expertise to identify vulnerabilities in code, community knowledge in the form of publicly available vulnerability databases could come to their rescue. For example, the Common Vulnerabilities and Exposures Database (CVE) contains data about already reported weaknesses. However, the support with available examples in these databases is scarce. CVE entries usually do not contain example code for a vulnerability, its exploit or patch. They just link to reports or repositories that provide this information. Manually searching these sources for relevant information is time-consuming and error-prone.**
In this paper, we propose a vulnerability detection approach based on community knowledge and clone detection. The key idea is to harness available example source code of software weaknesses, from a large-scale vulnerability database, which are matched to code fragments using clone detection. We leverage a clone detection technique from the literature, which we adapted to make it applicable to vulnerability databases. In an evaluation based on 20 reports and affected projects, our approach showed good precision and recall.

Security; Code Clones; Information Systems

I. Introduction

In today's interconnected world, security is one of the most important challenges for companies and institutions [16]. Vulnerabilities in a software system allow hackers to intrude and maliciously alter its behavior. The impact can range from minor, such as bypassing the copyright of a movie, to major, such as the malicious intrusion into a control system of a nuclear reactor [4]. An example for the latter case is the Stuxnet worm, which used a weakness in a vendor driver library to infect 100.000 systems worldwide and inflict physical damage. Consequently, organizations begin assigning a higher priority to security as a quality attribute in software development.

A key challenge is to check the complete source code for vulnerabilities to avoid the associated exploits. Since software developers are not equipped with the required security expertise, ideally, security experts should review the whole source code of a project. However, in the face of realistic projects that often include hundreds of thousands of lines of code, a manual check of the project by security experts is infeasible. To the rescue may come available community knowledge from vulnerability databases, such as the Common Vulnerabilities and Exposures (CVE) [18]. The CVE provides detailed knowledge about a large number of reported security flaws, including their impact. Developers may want to leverage this knowledge by detecting instances of the flaws in their projects. Unfortunately, the associated manual process is time-consuming and error-prone: Developers have to use a search engine in order to find relevant entries based on the names of the used libraries. Then they must manually scan the source code to uncover problematic uses of the affected libraries. Even worse, support with available examples in these databases is scarce. CVE entries usually do not contain an example exploit or patch, but just a link to a report or to a repository that provides additional information on proof of concepts, patches, and exploits.

In this paper, we address the following research question: *How can we harness available community knowledge to facilitate the detection of security vulnerabilities in software code?* We present an approach that uses *code clone detection* to detect instances of known vulnerabilities in source code. Clone detection aims to locate exact or similar code snippets, called *clones*, in or between software systems [21].

Our approach, illustrated in Fig.1, involves on a *security code repository* that contains security-relevant code snippets. Each snippet instantiates a known vulnerability.

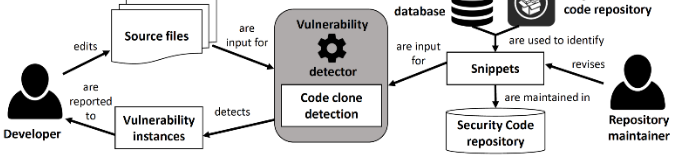


Figure 1. Approach Overview

The security code repository is created in a semi-automated process using automated searches over the CVE database and a large-scale code repository such as GitHub [6], and manual

DOI reference number: 10.18293/SEKE2019-183

refinement by developers. We detect duplicates of entries by using clone detection. To this end we have adopted an existing technique called SourcererCC [19], which fulfills two main prerequisites of our approach: It is efficient, as it scales to huge code bases with 100K LoC, and language-independent, as it supports arbitrary programming languages.

Our contributions are as follows:

- A *vulnerability detection technique* that uses and adapts an existing clone detection technique in order to detect security vulnerabilities, based on the given security code repository (Sec. III).
- A process for creating a *security code repository* with code snippets that instantiate known vulnerabilities, together with an initial version of such a repository (Sec. IV).
- An evaluation, in which our approach was able to detect the considered vulnerabilities with high precision and recall (Sec. V).

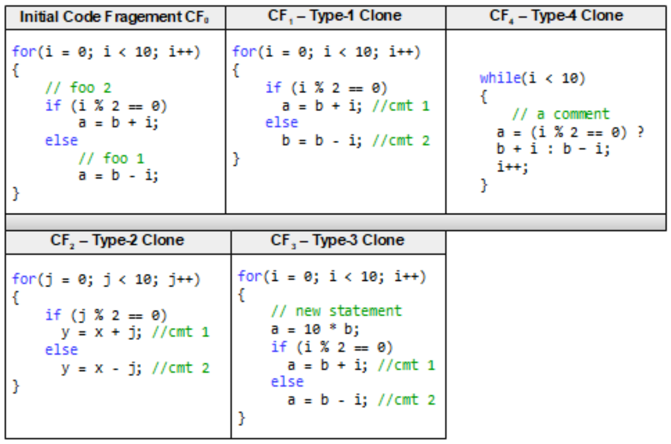
Clone detection has been used during vulnerability detection before. However, previous approaches were mostly limited to a particular programming language and suffered from scalability issues. We discuss related work in Sec. VI.

II. Background

We recall a common taxonomy of code clones and its implications for security.

Clone types. The common taxonomy of code clones [9] distinguishes four clone types, based on the degree of similarity: *Type-1 clones* are code fragments that are accurate copies of each other, excluding whitespaces, blank lines, and comments. *Type-2 clones* are structurally identical code fragments that may differ in the names of variables, literals and functions. *Type-3* or *near-miss clones* are syntactically similar code fragments that, opposed to Type-1 and Type-2 clones, may include changes like added or removed statements. *Type-4* clones are code fragments with a different syntax, but similar semantics. The example in Fig. 2 shows a code fragment CF_0 together with each clone type.

Security considerations. The clone types have different implications for security vulnerabilities. A vulnerability in a code fragment most likely also affects Type-1 clones of that fragment, since in most programming languages white spaces, blank lines and comments do not change the behavior. Neither does the change of variable names in Type-2 clones. However, the change of literal and method names can have an impact: a vulnerability may only occur when a specific method is called or when specific literals are used. Type-3 are particularly challenging for our approach, since an added line may render an insecure fragment secure, and vice versa. For example, consider the infamous buffer overflow weakness, where the problem is that the buffer size is not checked before writing or reading of it. A range check before accessing the buffer would fix this error, but the resulting code fragment is still a Type 3 clone. Type-4 clones regard the semantics of code snippets. The security impact of these clones depends on the chosen semantic representation. The typical means for checking semantic equivalence (such as pre- and post-conditions) are orthogonal to contained security vulnerabilities. Therefore, we do not consider Type 4 clones in our approach.



Initial Code Fragement CF_0

```
for(i = 0; i < 10; i++)
{
    // foo 2
    if (i % 2 == 0)
        a = b + i;
    else
        // foo 1
        a = b - i;
}
```

CF_1 – Type-1 Clone

```
for(i = 0; i < 10; i++)
{
    if (i % 2 == 0)
      a = b + i; //cmt 1
    else
      b = b - i; //cmt 2
}
```

CF_4 – Type-4 Clone

```
while(i < 10)
{
    // a comment
  a = (i % 2 == 0) ?
  b + i : b - i;
  i++;
}
```

CF_2 – Type-2 Clone

```
for(j = 0; j < 10; j++)
{
    if (j % 2 == 0)
      y = x + j; //cmt 1
    else
      y = x - j; //cmt 2
}
```

CF_3 – Type-3 Clone

```
for(i = 0; i < 10; i++)
{
    // new statement
    a = 10 * b;
    if (i % 2 == 0)
      a = b + i; //cmt 1
    else
      a = b - i; //cmt 2
}
```

Figure 2. Clone-Types 1 to Type 4

III. Vulnerability Detection

Our vulnerability detection approach involves four steps: *Pre-Processing*, *Code Processing*, *Clone Detection* and *Results*. The Pre- and Code Processing consists of the substeps Parsing & Tokenizing and Indexing. Only the Pre-Processing also contains the step of the CVE Data linking to enrich the code snippets with meta-information out of the CVE.

The input for the Pre-Processing step are the vulnerable code snippets of the security code repository including their assigned CVE metadata. During parsing, we identify contained methods and constructors of each source file. Within the tokenization, we create for each found method and constructor a separate token file containing the occurred tokens. Furthermore, for each of these files a file with bookkeeping information, in particular the CVE id to later query concrete CVE details, will be created. They also consist of links to code fragments which represent an example patch and their exploit to give developers a better understanding of the weaknesses for patching them afterward. The resulting tokens will be indexed and are the outcome for the preprocessing as well as the bookkeeping CVE information. It has to be applied once each time if the content of the code repository changes.

The Code Processing takes place every time if the source code has been changed. Thereby, the same Parse & Tokenizing and indexing like in the pre-processing will be applied but without adding CVE data to the bookkeeping information. The output of this step are the indexed tokens of source code files.

During the clone detection phase for each code fragment of the security repository and for each method as well as constructor inside of source files will be analyzed whether there are code clones. In detail, the tokens of methods and constructors will be compared. If a match is found, then the checked source code is a code clone of an insecure code fragment, which implies that it potentially also contains a security flaw. These code clones will be interleaved with the CVE data out of the bookkeeping information, which are the results of the vulnerability detection. Thus, should help developers to receive more knowledge to patch insecure source code fragments. The described approach is visualized in Fig. 3. Later in this chapter, the clone detection will be described in detail.

The effectiveness of this approach relies among others on the data of the reference repository. Therefore, it is inevitable to ensure the adaption and enrichment of knowledge by developers or a repository maintainer. They are able to add new vulnerable, patch and exploit code and modify already stored data.

A big problem for the code clone detection is the time complexity to compute the pairwise similarity for each code fragment combination. Execution time majorly scales with the size of the input precisely of the number of lines of code (LOC) that are processed and searched. For code clone detectors it is prohibited as a time complexity of scalability - $O(n^2)$. If the granularity of the code clone detector is method based, the similarity comparisons increase quadratically with the number of methods. For the SourcererCC various heuristics for reducing the number of similarity computations are described by Sajnani et al. [19]. In comparison of their approach to other state-of-the-art code clone detectors like CCFinderX [8], Deckard [29], iClones [30] and NiCad [31] they reach almost the same time complexity of inputs less than one million LOC. For all bigger input sizes, the SourcererCC has the best execution time. Furthermore, the SourcererCC is the only clone detector of the competing tools that scale to large input sizes of 100 million LOC and is able to consider type 1 to 3 clones.

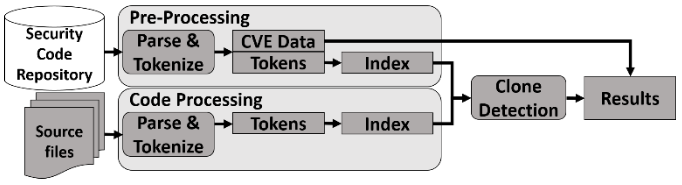


Figure 3. Vulnerability Detection Process

SourcererCC. As a basis for clone detection, we have adopted a state-of-the-art code clone detector named SourcererCC [19]. The detector supports Type 1 to 3 clones and scales to large-scale project repositories while providing high precision and recall. To quantitatively infer if two code snippets are clones a similarity function is applied which returns the non-negative degree of similarity between two code snippets. The higher the value of similarity, the bigger is the likeness between them. This function includes a threshold value ϑ that identifies the lower-bound of the similarity value from which two code fragments count as code clones. In other words, it is a percentage value that represents how many tokens at least should be shared by two code fragments to be identified as code clones. This similarity value is the output of the clone detection process. In the following, the similarity measurement is described formally:

Given two projects P_x and P_y, f as similarity-function and ϑ as threshold, the aim is to find all code block pairs $P_{x,B}$ and $P_{y,B}$ such that $f(|P_{x,B}|, |P_{y,B}|) \geq \lceil \vartheta * \max(|P_{x,B}|, |P_{y,B}|) \rceil$.

Adaptation of SourcererCC. We performed two main adaptations of SourcererCC for our approach:

First of all, we adapted the format of the token files to our needs and implemented a suitable tokenizer; thus its resulting token files will be interleaved with related information out of the CVE Database. Secondly, we adopted the code clone detector itself to consider only inter-project clones, as discussed later in this section. The main reason for this is that we plan to find clones between an external code repository and a project and not within a single project, like the clone detection is often used for.

Tokenizer. SourcererCC comes with a tokenizer for Java, C and C#. However, for our approach, we needed a method to interleave tokens with CVE meta-information such that clone detection results could provide further security information. Therefore, a custom token format was designed that combines the token information with the additional bookkeeping details, including the CVE id and the metadata of vulnerabilities. This information allows us to trace back from code fragments to the underlying weakness.

To apply the tokenizer a parsing of the source files is needed, such that the tokenizer gets software artifacts of source files like method names and constructors. For exemplary apply and evaluate our approach, we focus on java source code such a java parser is used for parsing code. For each method or constructor, a new list of tokens is created. Whitespaces, operators and comments are ignored. During this procedure, the tokenizer loads the needed metadata out of the security repository. The output of our tokenizer are two files, one including the tokens of a code fragment and the other with the described bookkeeping information enriched with security knowledge.

For further illustration, we present an example method before the preprocessing was applied in Fig. 4.

```
public class TokenizerExample
{
        private boolean isEven(int n)
        {
        int two = 2;
        return n % two == 0;
        }
}
```

Figure 4. Example: Java Input Source Code for Tokenizer

Figure 5 presents the output of the tokenization of Fig.4.

```
{0,private,2,boolean,isEven,int,two,n,return}
```

Figure 5. Created tokens of Fig. 4

To complete the tokenization, we count the number of appearances of each token and add them to the output. Hence, a token-file consist of every occurred method name, variable name and its datatypes as well as return values that are named and counted. Regarded to the design constraint for the Java and C# languages, all executable code must be contained inside of method bodies. Therefore, the granularity level for the implemented tokenizer is set to method-based tokenization. This means that only method respective constructors will be processed inside of a class. Imports and class names will be ignored. For other languages like C and C++ it is necessary to adapt the tokenizer to a class-based tokenization through the absence of this design constraint.

Inter-project clones. Clones can be either intra- or inter-project clones, meaning that the instances of a clone may come either from the same project or from different ones. In our approach, we are only interested in inter-project clones, since we aim to find matches between a given project source code and our security code repository. More formally, let A be the set of source files from the input project and B the set of snippets from the security code repository. Furthermore, let (a_0, …, a_n) and (b_0, …, b_n) with $n,m \in \mathbb{N}$ code fragments, for which $a_i \in A$ and $b_j \in B$ with $i,j \in \mathbb{N}$. We are interested in pairs of code fragments (a_i, b_j) being code clones.

SourcererCC finds both intra- and inter-project clones, and by default it does not provide a configuration option to deactivate the detection of intra-project clones. Therefore, the *Pre-Processing* phase was added and the inputs for the clone detection was adapted to ignore intra-project clones. Figure 3 presents the modified behavior of the clone detector.

IV. Security Code Repository

In our approach, a prerequisite for vulnerability detection is a reference code repository called *security code repository*. This repository contains source code snippets that instantiate already reported weaknesses. Each snippet consists of example code for a vulnerability, its exploit and a patch. We now describe the procedure for creating a security code repository, which we used to test and evaluate our technique. Our process relies on a large-scale repository in which such snippets can be found. To this end, we used GitHub, a suitable repository that is freely accessible to the public. Our process, shown in Fig. 6, contains the four steps *Extract*, *Search*& *Filter*, *Export*, and *Proof*.

Extract. To find security-related code snippets on Github, adequate terms for the search are needed. We extract them from the CVE database. A possible way to identify a concrete vulnerability is its unique CVE identifier [18]. Therefore, we extract these CVE-ids for the search to find security issues.

Search and Filter. For searching vulnerabilities on Github, the pre-extracted CVE-ids will be used. If a file or a commit within a project matches these identifiers, they will be considered for further processing. To adjust the results of the search to specific programming languages, we defined a file type filter. For example, to restrict the search to Java files, the file endings will be checked for the tag *.java*. Not every content of the found files is security-related. Therefore, a manual prove is necessary to ensure that resulted files contain only security-related code. This procedure is described later in this section.

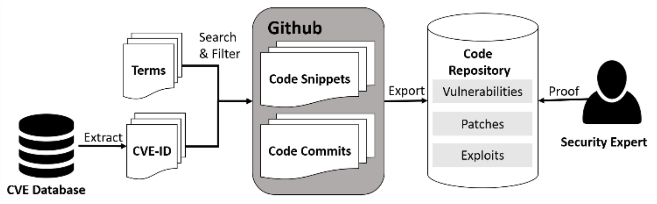


Figure 6. Repository Creation for Reference Code Fragments

Classification. Furthermore, we defined terms to distinguish between the three classes of security-related files; vulnerabilities, exploits and patches. For this classification, we use a simple check whether terms are substrings of texts inside of commits or project descriptions. Examples terms for the class *patches* are fix, solve, update, patch for *vulnerabilities* the term vulnerability and for *exploits*, the substrings are proof of concept (POC) and exploit. We retrieved these terms by the manual unsystematic analyze of founds of the CVE id search received from GitHub. The founds will be automatically classified by these terms into the three mentioned classes.

Export. The classified founds are stored into a code repository with a SQLite database inside of its root. For all matches its related and stored meta-information inside of the CVE will be extracted into the SQLite database. This meta-information are for example the concrete CVE description, the CVE-id and a scoring which represents their characteristics, impact and severity. The vulnerability scoring is based on the Common Vulnerability Scoring System (CVSS) [28].

Proof. As post-processing the repository content has to be manually reviewed to exclude file parts that do not contribute to a vulnerability. If a file with a vulnerability in it is found, often only a part of the file represents a vulnerable code snippet. A security expert has to delete the unaffected methods in these files such that only the critical code remains.

For the creation of a code repository, 20 different weaknesses out of the CVE database were selected. Our previous explained semi-automated tool-based approach found source code examples of exploits, vulnerabilities and its patches assigned to CVE-ids. Thus, security-related code fragments are extracted out of Github. We have reviewed a subset of 102 reported security flaws to check their suitability for representing a vulnerability code snippet that is usable for the approach described within this work. For example, not eligible vulnerabilities can be patched by only importing a newer library version or loading them dynamically by a string literal. Furthermore, code fragments for which weaknesses are spread over multiple methods were partially also ignored. The reason for this is that in the most cases code changes were too small to match the three different types of code clones meaningfully. Table 1 shows the selected vulnerabilities with their associated CVE, their scoring and the affected product, as it is stored into the CVE Database. The created security code clone repository including the SQLite database is uploaded to Github [24].

V. Evaluation

We evaluated the suitability of the described approach to address our initial research question: *How can we harness available knowledge to facilitate the detection of security vulnerabilities in software code?* To this end, we consider the following evaluation research questions:

RQ1: How many of the vulnerabilities inside of the CVE are attributed to source code?

Not for every vulnerability the reason is a weakness in source code. This question should identify how valuable the focus on source code vulnerabilities is and how many of them could be

TABLE I. CODE REPOSITORY WITH SECURITY-RELATED CONTENT

CVE	Scoring	Product
CVE-2006-2806	7.8	Apache Java Mail Enterprise Server
CVE-2007-5461	3.5	Apache Tomcat
CVE-2007-6203	4.3	Apache HTTP Server 2.0.x and 2.2.x
CVE-2008-2086	9.3	Sun JDK
CVE-2008-5515	5.0	Apache Tomcat
CVE-2009-2693	5.8	Apache Tomcat
CVE-2009-2901	4.3	Apache Tomcat
CVE-2010-4172	4.3	Apache Tomcat
CVE-2011-1475	5.0	Apache Tomcat
CVE-2011-3190	7.5	Apache Tomcat
CVE-2011-3377	4.3	Ubuntu Linux, Opensuse, Redhat Icedtea-Web
CVE-2012-1621	4.3	Apache Open For Business Project
CVE-2012-2459	5.0	Bitcoind und Bitcoin-Qt
CVE-2016-3897	4.3	Google Android
CVE-2016-6723	5.4	Google Android
CVE-2017-0389	7.8	Google Android
CVE-2017-0846	5.0	Google Android
CVE-2017-1217	4.3	IBM WebSphere Portal
CVE-2017-1591	4.3	IBM DataPower Gateway
CVE-2017-9096	6.8	iTextpdf iText

found through using the knowledge of source code of entries stored inside of the CVE by applying the described approach.

RQ2: How accurate is our approach at detecting previously reported vulnerabilities?

We want to check whether it is possible to identify source code reasoned weaknesses through a subset of the reported vulnerabilities stored inside of publicly accessible databases like the CVE.

RQ3: How well does our approach distinguish between vulnerable and patched code fragments?

Sometimes only the change of a few lines of code is necessary to remove a security flaw inside of a code snippet. We investigate on which granularity we can distinguish between patched and insecure code fragments through clone detection.

A. *RQ1: How many of the vulnerabilities inside of the CVE are attributed to source code?*

First, we investigate how feasible it is to use the information of known and documented vulnerabilities reported in databases like the CVE. For this proof, we check the ratio of entries that could be retrieved through errors within source code to them which do have other origins. The more weaknesses based on source code, the better is the concentration on detecting security flaws within code artefacts. Through this survey the capability of using the content of the publicly accessible database CVE to recognize security issues invoked through source code will be validated. To apply this investigation, the non-code content of the CVE was manually screened. We recognized that in August 2018 only for 62 % of the 103745 CVE entries, a Common Weakness Enumeration (CWE) identifier is assigned.

The CWE compresses different types of weaknesses, which are all identified through a unique CWE-id and categorize different manifestations of vulnerabilities. An example of these types is *CWE-306: Missing Authentication for Critical Function*. All to this type associated CVE entries are vulnerabilities because of the absence of authentication for critical functions. Therefore, CWE-ids are a well-suited attribute found with our approach. A problem is that there is no identifier for a type that implies all security issues appear within source code. For every set CWE type, it was systematically proved whether it is possible to retrieve out of its description the origin they belong to; induced by source code or others like configurations. A further problem is the absence of assigned CWE-ids for 38 % of the entries of the CVE. The besides without any type information were investigated with the use of their description. Manual checks show that it is possible to find CVEs based on source code via checking their descriptions for the occurrence of substrings. The used substrings were divided into five groups: File Name Endings, Attack Strategies, Configurations, Rejects and Unclassified.

The group File Name Endings contains substrings that represents the data types of programming language source files. The hypothesis is that if a concrete file is addressed within the description of a CVE than their occurrence is provable within source code. Examples of file endings are *.java, .cpp, etc.*

As attack strategy count for example *Cross-Site Scripting (XSS), Buffer Overflow etc.* The idea is to check for the names of attack strategies that maliciously uses a vulnerability occurred in source code like the buffer overflow example, which could be prevented to buffer length checks.

Configuration related vulnerabilities are identified through the occurrence of terms like *config, cfg etc.* We assume that for CVE descriptions that mention at least one of these terms, their belonging weakness is not detectable via source code analysis.

Rejects are the vulnerabilities that are still remain in the CVE database but were rejected after review. They are marked with the term *Rejected*. This are entries, which could be duplicates or not representing a vulnerability at all.

The left CVE entries that not belong to one of the mentioned groups were assigned to the group Unclassified CVE entries. They will be not considered to answer this research question. The partition of the CVE is summarized in a pie chart in Fig. 5.

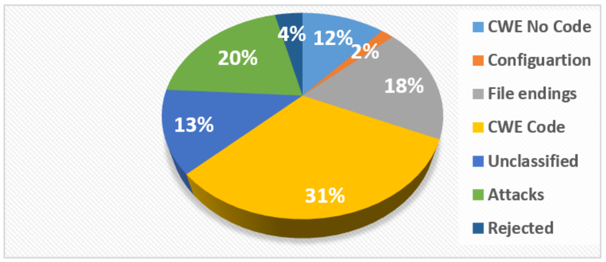


Figure 7. Classification of CVE Database Content

To conclude the results, it is shown that 69 % of the public available vulnerabilities are induced to source code issues. This is composed through the 31% of entries with CWEs that describes security flaws detectable within source code artefacts, the 28 % with a description containing file endings of source code files and the 20 % of CVEs that contain in their description terms of attack strategies uses security flaws within source code. To respond RQ1.1, it is a well-suited strategy to focus on source code for identifying known and reported weaknesses.

B. *RQ2: How accurate is our approach at detecting previously reported vulnerabilities?*

Experimental Setup

As vulnerability selection for our evaluation, we use the code repository described into Sec. IV. To ensure that not only Type-1 clones are detected but also Type-2 and Type-3 clones we modified each vulnerable code fragment to match the corresponding clone types. For example, all variable names were renamed for Type-2 clones. To obtain Type-3 clones, we removed or added some void statements to Type-1 and Type-2 clones. Our evaluation based on metrics of the information retrieval like *Recall*, *Precision* and F_1-*measure* as described by Manning et al. [15]. We measured the recall based on the chosen vulnerabilities considering the three different types of clones. In this case, we know exactly the number of weaknesses so that the recall can be measured precisely. The precision was measured by a manual validation of the found and highlighted security code clones. To combine precision and recall, we used F_1-measure.

The underlying clone detector uses a similarity function inside its clone detection process. This function can be configured by the threshold value ϑ. Three different ϑ values were used for the evaluation in combination with the three different types of clones. That means that for every clone type exists three iterations with different thresholds. Beforehand we examined distinct thresholds by hand. On the one hand, lowering the threshold could harm precision. On the other hand, a too high threshold could harm recall. Therefore 3.0, 5.5 and 8.0 were selected as values for ϑ.

Results and Discussion

Each iteration passes every vulnerability with the given configuration values. The Security Code Clone Detector has flagged all Type-1 clones for the three different ϑ values. That means that precision, recall, and F_1 are 100 % for Type-1 clones. The detection rate of Type-2 clones revealed that only eight security flaws out of 20 were detected with a ϑ of 8.0. The reason is that through the high ϑ value two code fragments must be too similar to be recognized as code clones. Due to this the higher the value of ϑ is the bigger the number of similar tokens inside of two code snippets have to be. Thus lead to the low recall for Types-3 clones cause the clones are too different to be code clones of each other.

In contrast, the precision always is 100 %. This could be attributed to the fact that we only focus on finding the three different types of security code clones. The set of data only consist of the given vulnerabilities and not of any secure code fragments. Table 2 summarizes the clone detection results for the security flaws and patches.

TABLE II. EVALUATION RESULTS

	Type-1 Clones				Type-2 Clones				Type-3 Clones				Fixes			
θ	TP	R	P	F_1	TP	R	P	F_1	TP	R	P	F_1	TP	R	P	F_1
3.0	20	100	100	100	20	100	100	100	20	100	100	100	1	100	5	9.52
5.5	20	100	100	100	20	100	100	100	12	60	100	75	3	100	15	26.08
8.0	20	100	100	100	8	40	100	57.14	0	0	100	0	11	100	55	70.97

In the result table TP stands for true positives, R means recall, P means precision and F_1 stands for F_1-measure. The high recall values for all code clone types and ϑ modifications show that we are able to recognize known and reported weaknesses via our approach. The results for the recall are interleaved with the size of the ϑ threshold. For small ϑ values every vulnerability inside of the reference repository is detected, but this leads to a lower precision. Therefore, the answer for RQ2 is that the vulnerability detection for the described approach performs very well with a leak of precision for Type-3 clones.

C. *RQ3: How well does our approach distinguish between vulnerable and patched code fragments?*

Experimental Setup

In the next step, we have focused on checking precision. Thereto the patched code fragments for the given vulnerabilities were used to examine which security flaws will be detected as vulnerable code clones falsely. In this case, we did not modify the patched code fragments to match each of the three different types of clones but used the code fixes directly as input for the detection process. Patched code fragments distinguish to their vulnerable complement with some changes that removes the weak parts of that fragment. These changes differ in complexity. Some have an extent of multiple lines but other distinguish only by a single line to their vulnerable counterparts. The ϑ configuration setup for every iteration was the same we mentioned in RQ2.

Result and Discussion

As described in Sec. III, Type-3 clones are hard to detect. The low precision values summarized in Tab. II can be ascribed to the few code modifications inside the patches. Fundamentally, the patches represent Type-3 clones. If a patch has enough code modifications regarding its vulnerability, we can identify it as secure code correctly. If the code changes are too small, it may be flagged as weakness falsely. Relating to RQ3, it is possible to distinguish between vulnerable and patched code fragments as far as enough code modifications are present. On the one hand, the capability to detect code clones of Type-3 increases the amount of secure code, which is falsely identified as insecure. This reduces the precision of the described approach. On the other hand, through code clones of Type-3, the possibility exists to find more code clones of vulnerable code fragments, which increases the recall. Hence our goal is a high recall during detection. Therefore, we accept the false positives within the patch detection.

D. *Threats to Validity*

For the validity check of our evaluation, we consider the types of threats to validity for empirical software engineering research defined by Wohlin et. al. [26].

Conclusion: Maybe the selection criteria for building the test set influence the results in terms of recall, precision, and F_1-measure. To be more precise, it is possible that the size of selected code fragments influence the capability to distinguish between patched and weak code fragments. We have not considered the size of needed patches to close security flaws.

Internal: The used Java Runtime Environment version and the library version of used source code are not considered. Some code fragments are only insecure with specific Java versions or library versions and are secure within other versions. This could result in false positives for the classification of source code fragments as vulnerable that are secure with the used versions. Furthermore, it is imaginable that there are other code clone detection approaches, which tackles the problem of vulnerability detection inside of code fragments better than the chosen one.

Construct: The configuration and the ϑ adjustment of the code clone detection approach affect the effectiveness of the described procedure. Furthermore, the workflow and the granularity level of the tokenizer influences the results of the code clone detection approach.

VI. Related Work

Vulnerability detection using clone detection. The clone detector ReDeBug [7] is language agnostic and uses a syntax-based pattern matching approach. It can detect some Type-3 clones but cannot detect Type-2 clones respectively clones with slight code modifications. Furthermore one of the design goals was a low false positive rate which harms recall. VulPecker [12] is a system for automatically detecting if a piece of software contains a vulnerability. It consists of a learning phase and a selection algorithm to identify vulnerabilities. This approach can detect Type-1, Type-2 and some Type-3 clones [13] in C/C++ code. CLORIFI [11] combines static and dynamic analysis to detect code clone vulnerabilities. It identifies the security code clones of known vulnerabilities with an n-token algorithm. With the help of concolic testing, CLORIFI tries to reduce false positives by verifying the security flaws. Our work is mostly complementary to the presented ones, as we focus on establishing a right balance between precision and recall, and use a state-of-the-art back-end clone detector that allows us to address scalability and language-independence simultaneously.

Static analysis for security. Our approach can be considered as a static analysis technique that can uncover vulnerabilities without executing the application at hand. Such techniques have been used successfully to uncover vulnerabilities, in some cases better than dynamic techniques such as penetration testing [20]. Most previous techniques are geared to detect specific vulnerabilities based on hard-coded solutions, such as SQL injections [14] and buffer overflows [27]. Fischer et al. [5] have used static analysis to study how severely Android apps are affected by vulnerabilities resulting from copying code snippets from StackOverflow examples. In our previous work [25] we present a tool-based approach that scans imported Java dependencies for known vulnerabilities. It checks the CVE if an entry for the corresponding library exists. Furthermore, we present an approach [3] for maintaining a knowledge base of security knowledge that is used to keep co-evolve the system design after changes in the availability knowledge (e.g., an encryption algorithm previously deemed as secure is broken). However, this work was focused on the design model level rather than on code-level vulnerabilities.

Code clone detection. Different code clone detection approaches with distinct capabilities exist. In their survey, Sheneamer et al. [21] distinguish the following main classes of approaches. Text-based techniques (e.g. [1]) compares the similarity of code fragments based on terms of textual content. While focusing mostly on Type-1 clones without preprocessing, they are language-independent and easy to implement. Lexical techniques (e.g. [8,19]) divide the input code into a sequence of tokens that are converted into token sequence lines, which are then matched to another. Such techniques can detect various code clone types with higher recall and precision than text passing techniques. Syntactic techniques are either metric-based or tree-based. Tree-based (e.g., [2]) and graph-based techniques (e.g., [22]) parse the code into abstract syntax trees and find cloned code parts using tree-matching algorithms. Metric-based ones (e.g. [17]) compare source code snippet based on metric vectors created for each code snippet. This technique is able to detect Type-1 and Type-2 clones with high time complexity. It reduces the complexity of text-based approaches. Semantic techniques (e.g., [10]) detect two fragments of code that perform the same computation but have differently structured code, which was already named as Type-4 clone.

VII. Conclusion

The use of a single vulnerable code snippet can make a whole system insecure [23]. We introduce an approach to prove code fragments of reported vulnerabilities automatically. We detect insecure parts of source code via code clone detection by using code fragments that contain known vulnerabilities. As a result, a developer can be supported in the writing of secure code right from the beginning. With the aid of a database, it is possible to show information about reported security flaws. This information can be used to understand a weakness and, if necessary, provide a patch. The results of the evaluation show that detecting insecure code fragments work very well. We show that there are difficulties in distinguishing between patched and vulnerable source code fragments that are too similar to each other. The capability to detect small differences between them depends on the configuration and the distinction of line circumference from the vulnerable and patched fragments. Our work provides an *Eclipse* plug-in for supporting Java software developers [24]. Furthermore, we investigated the feasibility to distinguish between patched and vulnerable code snippets for our approach. We analyzed the CVE database content to underpin leveraging knowledge of reported vulnerabilities for the detection of security flaws within the source code.

Our future research is striving to compare artificial intelligence approaches in the form of neural networks with the static code clone detection approach described in this work. To solve the problem of our internal validity, we want to enhance the described approach such that the JRE Version of software projects and the recognition of library versions will be considered. Furthermore, a user study about the perception and effectiveness of helping developers and security experts to classify source code fragments is planned. The capability of the code clone detectors in the described approach relies on the data included in the reference code repository. Therefore, we expect to investigate further techniques to enhance the security-related source code within the code repository via leveraging community knowledge. Furthermore, we plan to improve the semi-automatically classification into exploit, vulnerable and patch code fragments.

VIII. ACKNOWLEDGMENT

This work was supported by the German Research Foundation (DFG) under SecVolution (2016 – 2019).

REFERENCES

[1] Baker, B.S.: On finding duplication and near-duplication in large software systems. In: Reverse Engineering, 1995., Proceedings of 2ndWorking Conference on. pp. 86{95. IEEE (1995)
[2] Baxter, I.D., Yahin, A., Moura, L., Sant'Anna, M., Bier, L.: Clone detection using abstract syntax trees. In: Software Maintenance, 1998. Proceedings., International Conference on. pp. 368{377. IEEE (1998)
[3] Bürger, J., Strüber, D., Gärtner, S., Ruhroth, T., Jürjens, J., Schneider, K.: A framework for semi-automated co-evolution of security knowledge and system models. In: Journal of Systems and Software 139, pp. 142{160 (2019)
[4] Devanbu, P.T., Stubblebine, S.: Software engineering for security. In: Finkelstein, A. (ed.) Proceedings of the Conference on The Future of Software Engineering. pp. 227{239. ACM, New York, NY (2000)
[5] Fischer, F., Böttinger, K., Xiao, H., Stransky, C., Acar, Y., Backes, M., Fahl, S.: Stack overow considered harmful? the impact of copy&paste on android application security. In: Security and Privacy (SP), 2017 IEEE Symposium on. pp. 121{136. IEEE (2017)
[6] Inc., E.: Github (2008), https://github.com/
[7]. Jang, J., Agrawal, A., Brumley, D.: Redebug: finding unpatched code clones in entire os distributions. In: Security and Privacy (SP), 2012 IEEE Symposium on. pp. 48{62. IEEE (2012)
[8] Kamiya, T., Kusumoto, S., Inoue, K.: Ccfinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering 28(7), 654{670 (2002)
[9] Koschke, R.: Survey of research on software clones. In: Dagstuhl Seminar Proceedings. Schloss Dagstuhl-Leibniz-Zentrum für Informatik (2007)
[10] Krinke, J.: Identifying similar code with program dependence graphs. In: Reverse Engineering, 2001. Proceedings. EighthWorking Conference on. pp. 301{309. IEEE (2001)
[11] Li, H., Kwon, H., Kwon, J., Lee, H.: Clorifi: software vulnerability discovery usingcode clone verification. Concurrency and Computation: Practice and Experience 28(6), 1900{1917 (2016)
[12] Li, Z., Zou, D., Xu, S., Jin, H., Qi, H., Hu, J.: Vulpecker: an automated vulnerability detection system based on code similarity analysis. In: Proceedings of the 32nd Annual Conference on Computer Security Applications. pp. 201{213. ACM (2016)
[13] Li, Z., Zou, D., Xu, S., Ou, X., Jin, H.,Wang, S., Deng, Z., Zhong, Y.: Vuldeepecker: A deep learning-based system for vulnerability detection (2018)
[14] Livshits, V.B., Lam, M.S.: Finding security vulnerabilities in java applications with static analysis. In: USENIX Security Symposium. vol. 14, pp. 18{18 (2005)
[15] Manning, C.D., Raghavan, P., Schtze, H.: Introduction to Information Retrieval. Cambridge University Press, Cambridge, United Kingdom (2009)
[16] Mayer, C.P.: Security and privacy challenges in the internet of things: 158 kb / electronic communications of the easst, volume 17: Kommunikation in verteilten Systemen 2009 (2009)
[17] Mayrand, J., Leblanc, C., Merlo, E.: Experiment on the automatic detection of function clones in a software system using metrics. In: icsm. vol. 96, p. 244 (1996)
[18] Mitre: Common vulnerability and exposures (1999), https://cve.mitre.org/
[19] Sajnani, H., Saini, V., Svajlenko, J., Roy, C.K., Lopes, C.V.: SourcererCC: Scaling Code Clone Detection to Big-Code. In: 2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE). pp. 1157{1168 (May 2016)
[20] Scandariato, R., Walden, J., Joosen, W.: Static analysis versus penetration testing: A controlled experiment. In: Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on. pp. 451{460. IEEE (2013)
[21] Sheneamer, A., Kalita, J.: A Survey of Software Clone Detection Techniques. International Journal of Computer Applications 137(10), 1{21 (2016)
[22] Strüber, D., Acreţoaie, V., Plöger, J.: Model clone detection for rule-based model transformation languages. Software & Systems Modeling 18(2), pp. 995{1016 (2019)
[23] US-Cert: United states computer emergency readiness team (2003), https://goo.gl/ZCuCc8
[24] Viertel, F.P., Brunotte, W., Strüber, D., Schneider, K.: Security Code Repository and Code Clone Detection Eclipse Plug-In (2018), https://github.com/dev-se/sccd
[25] Viertel, F.P., Kortum, F., Wagner, L., Schneider, K.: Are third-party libraries secure? a software library checker for java. In: The 13th International Conference on Risks and Security of Internet and Systems, CRISIS (2018)
[26] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M.C., Regnell, B., Wesslén, A.: Experimentation in software engineering. Springer Science & Business Media (2012)
[27] Zitser, M., Lippmann, R., Leek, T.: Testing static analysis tools using exploitable buffer overflows from open source code. In: ACM SIGSOFT Software Engineering Notes. vol. 29, pp. 97{106. ACM (2004)
[28] NIST: Common Vulnerability Scoring System (2005), https://nvd.nist.gov/vuln-metrics/cvss
[29] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In Software Engineering, 2007. ICSE 2007. 29th International Conference on, pages 96{105, May 2007.
[30] N. Gode and R. Koschke. Incremental clone detection. In Software Maintenance and Reengineering, 2009. CSMR '09. 13th European Conference on, pages 219{228, March 2009.
[31] J. R. Cordy and C. K. Roy. The nicad clone detector. In Proceedings of the 2011 IEEE 19th International Conference on Program Comprehension, ICPC '11, pages 219{220, Washington, DC, USA, 2011. IEEE Computer Society.