Hindawi
Wireless Communications and Mobile Computing
Volume 2022, Article ID 3822836, 12 pages
https://doi.org/10.1155/2022/3822836

Research Article

A Novel Vulnerable Code Clone Detector Based on Context Enhancement and Patch Validation

Junjun Guo, Haonan Li, Zhengyuan Wang, Li Zhang, and Changyuan Wang

School of Computer Science and Engineering, Xi'an Technological University, Xi'an, 710021 Shaanxi, China

Correspondence should be addressed to Junjun Guo; guojunjun@xatu.edu.cn

Received 12 March 2022; Revised 30 April 2022; Accepted 12 May 2022; Published 4 June 2022

Academic Editor: Jun Ye

With the rapid growth of open-source software, code cloning has become increasingly prevalent. If there are security vulnerabilities in a cloned code segment, those vulnerabilities may spread in the related software to potentially lead to security incidents. The existing methods of vulnerable code detection are performed on the condition that the source code is converted into an intermediate representation. However, these methods do not fully consider the rich semantic knowledge and patch information available for vulnerable codes, which can induce a high false positive rate (FPR). To address this problem, this paper proposes a vulnerable code clone detection method based on code fingerprints, named the Context-enhanced and Patch-validation-based Vulnerable code clone Detector (CPVDetector). A fingerprint database is built for functions, code snippets, and patches derived from preprocessed vulnerable source code. The target code to be detected is firstly transformed into function-level fingerprints. If clone detection fails at this coarse granularity, the detector is then applied at the finer line-level granularity. When fingerprint matching is successful between the target code and the vulnerable code segments, the detector will proceed to verify the context of vulnerable codes. Finally, CPVDetector can verify the fingerprints of patches corresponding to vulnerable codes to further reduce the FPR. Based on the generally accepted classification of code clones, CPVDetector can identify Type 1 and Type 2 vulnerable code clones at the coarse-grained level and offers significantly improved detection sensitivity for Type 3 and Type 4 code clones at the fine-grained level. Experimental results show that the proposed method can achieve high accuracy with a fast detection speed, and the FPR is as low as 2.35%, which is less than one-third of that of other existing methods. In view of its competitive performance and efficiency, CPVDetector can be applied in large-scale vulnerable code detection scenarios.

1. Introduction

The number of open-source software projects has increased rapidly in recent years [1, 2]. There were 170 million code repositories in GitHub [3] in 2020, of which 54.21 million were active code repositories, an increase of 36.4% compared to 2019. In the software development stage, programmers need to complete their tasks within a given time frame [4]. Therefore, copying and pasting of code often occur, either without any modifications or with only some simple modifications in the copied code segments, such as identifier replacement, sequence adjustment, or annotation modification. It is evident that such frequent copy-and-paste operations may be detrimental to software quality and maintenance [5–9]. The process of copying code snippets is called code cloning. Cloning vulnerable codes may cause the same vulnerabilities to be propagated in the development process, which can lead to security incidents [6].

Developers usually modify cloned code to meet software requirements by performing operations such as deleting unnecessary statements or adding some assertion statements for debugging. These operations may modify cloned vulnerable code, making vulnerability detection more difficult. Nevertheless, even if the structure of vulnerable code is modified, vulnerabilities may still exist in the cloned code segment [10, 11]. This kind of vulnerability is known as restructured clone vulnerability. Deckard, ReDeBug, and VUDDY can detect restructured clone vulnerabilities with extremely limited semantic information on the vulnerable code [12–15]. However, existing tools for restructured clone detection cannot validate associated patch files, which results in a high false positive rate [16–20].

This paper proposes the Context-enhanced and Patch-validation-based Vulnerable code clone Detector (CPVDetector) as a tool for vulnerable code clone detection based on code fingerprints, which can effectively identify common types of code clones. The target code is detected at function-level and line-level granularity. A code fingerprint consists of a series of MD5 hash bits with a length of 32 that is simple and unique. In addition, the proposed method can leverage the context of vulnerable code snippets to reduce false positives caused by context-sensitive vulnerability. Through patch validation, CPVDetector can effectively locate patch codes in the target segments; thus, it can further reduce the false positive rate. Accordingly, CPVDetector can effectively identify vulnerabilities in practical software projects that cannot be detected by other state-of-the-art methods. More importantly, the proposed method can reduce the FPR to as low as 2.35% while improving the F-measure by approximately 30% when compared to existing methods.

The main contributions of this paper are as follows:

(1) We collect 983 vulnerabilities in common C/C++ open-source software projects from 2010 to 2020 and build a scalable fingerprint database for vulnerable code

(2) We propose a 2-level vulnerable code clone detection tool based on the source code fingerprints of functions and statements. To further reduce false positives, the context and patch information of vulnerable code is considered in line-level detection

(3) CPVDetector can detect four different types of vulnerable code clones. Compared to existing code clone detection methods, the proposed detection method can achieve the best balance in terms of accuracy and speed

The rest of this paper is organized as follows. Section 2 describes the classification of code clones and the existing code clone detection methods. Section 3 describes the collection and processing of vulnerability data and fingerprint generation. Section 4 presents how CPVDetector performs vulnerable code clone detection. In Section 5, we evaluate the proposed CPVDetector in comparison with other vulnerable code clone detection methods. Finally, Section 6 presents the conclusion and future work.

2. Background

2.1. Code Clone Classification. The original codes in cloned code segments are usually modified through operations such as variable renaming, redundant code insertion, annotation modification, data type modification, operator modification, statement order modification, code block order modification, and equivalent conversion of control structures [21, 22]. A set of widely accepted definitions for the classification of code clones is given as follows [5, 6]:

Type 1: exact clone. The code layout may be modified by modifying spaces and tabs, and the annotations may be edited; however, the code part is copied without any modifications.

Type 2: renamed clone. In addition to the modifications of Type 1, only the data types of variables and function return values are modified, or identifiers and variables are renamed.

Type 3: restructured clone. In addition to the modifications of Type 1 and Type 2, structural modification operations such as deletion, insertion, and rearrangement of statements are performed to generate restructured clones.

Type 4: semantic clone. In addition to the modifications of Type 1, Type 2, and Type 3, although the semantics of the code do not change, the syntax is adjusted.

2.2. Related Works. The existing code clone detection methods can be divided into five types: text-based, token-based, graph-based, abstract-syntax-tree-based, and metric-based methods.

(1) *Text-Based Methods.* Text-based methods convert source codes into sequences of lines or segments. To find a similar sequence, a given code segment is compared with other segments. To improve the detection accuracy, it is necessary to perform a series of preprocessing steps on the source code, such as standardization and normalization [23]. With granularity at the level of lines of code, ReDeBug leverages the sliding window method on the source code to detect files by means of a Bloom filter. ReDeBug can effectively detect Type 3 code clones, but it cannot effectively detect Type 1 or Type 2 code clones; as a result, a large number of clones of vulnerable code are missed. In addition, the false positive rate is high for methods based on line-level detection that do not consider code context or patch knowledge. With granularity at the level of functions, VUDDY can effectively detect clones of vulnerable code by detecting the fingerprint of the code for each function. However, VUDDY has difficulty detecting some common methods of code modification, such as code word order changes or additions and deletions of redundant code. Based on the sequences obtained via compilation and decompilation of Java source code, a tree-based clone detector has been proposed that can effectively recognize Type 1, Type 2, and Type 3 clones [24, 25]. Jadon proposed a technique for detecting Type 3 clones and quantifying their similarity [26]. This method can identify Type 3 clones for the C language by means of intermediate representation vectors and a support vector machine classifier

(2) *Token-Based Methods.* The source code is parsed into a token sequence that is easy to compute. Taking CCFinder [27] as an example, the similarity between two token sequences is calculated using a suffix tree algorithm. However, CCFinder has a rather high false positive rate because of its abstraction and filtering heuristics. SourcererCC can detect Type 3 code clones by using token package technology [28]. If the similarity between two functions exceeds a predetermined

threshold, then a clone is detected. However, if a statement is inserted into vulnerable code to make a simple modification, for example, adding an if statement, then SourcererCC will not be able to accurately detect the code clone. Moreover, Nishi and Damevski applied adaptive prefix filtering heuristics in a clone detection method [29] that could find Type 1, Type 2, and Type 3 clones. Wang et al. proposed CCAligner, which is a token-based clone detection tool [30] that uses C and Java files as data sets and can detect Type 1, Type 2, and Type 3 clones

(3) *Tree-Based Methods*. The source code is represented as an abstract syntax tree, in which nodes represent program entities and edges represent the connections between these entities. A heuristic tree search algorithm is used to identify clone pairs in similar subtrees. For example, Deckard constructs abstract syntax trees for files and then extracts feature vectors from the tree. After the feature vectors are clustered based on the Euclidean distance, vectors that are sufficiently close to each other in the Euclidean space are identified as code clones. This method based on a tree structure incurs a considerable time overhead because the subgraph isomorphism problem is an NP-complete problem. In addition, Deckard has a relatively high false positive rate, which indicates that cloning is not necessarily present in vulnerable codes with similar abstract syntax trees. Yang et al. proposed an automatic code clone detection method [31] that generates an abstract syntax tree with function-level units and uses the Smith–Waterman algorithm to calculate the score for each function

(4) *Graph-Based Methods*. In these methods, a program is converted into a graph such as a program dependence graph (PDG). These methods can achieve a higher level of abstraction of their code representations than other methods because they consider the semantic information of the source code. Crussell et al. proposed a detection tool based on the PDGs of C-language source code [32]. This tool uses the locality-sensitive hashing (LSH) algorithm to search for vectors that approximate nearest neighbors and the Min-Hash algorithm to calculate the similarity [33].

(5) *Metric-Based Methods*. Various metrics of the source code are calculated, such as the number of lines of code, the number of operators, and the cyclomatic complexity. Then, these metrics are compared to detect clone pairs that have the same metrics. Svajlenko and Roy summarized the concept of CloneWorks [34], which is a Type 3 clone detection tool that uses the IJaDataset and the Jaccard similarity measure for clone detection

The code clone detection methods introduced above all have relatively high detection capabilities for Type 1 code clones. Text-based or token-based methods are better for Type 2 code clone detection. Tree-based or metric-based methods are suitable for detecting Type 3 code clones. Graph-based methods can detect some Type 4 code clones, but graph generation and subsequent detection are time-consuming processes. Singh proposed a hybrid method based on code metrics and PDGs to convert Java source code into abstract syntax trees and PDGs [35], which can detect Type 1, Type 2, and Type 3 code clones. VulPecker uses a variety of source code representation methods and similarity calculations for code clone detection [36]. However, due to the poor efficiency of this method, it is not suitable for code clone detection for large open-source projects.

3. Vulnerability Fingerprint Database

A series of preprocessing steps is performed on obtained sample data of vulnerable source code, and fingerprints are generated in accordance with granularities at the code line and function levels. Thus, a fingerprint database for code clone detection is constructed.

3.1. Data Collection. Samples of vulnerable source code are obtained from open-source projects in GitHub, including the Linux kernel, FFmpeg, and OpenSSL. Corresponding patch information is obtained from the submission histories. Vulnerable code segments are extracted from the diff files in the patches. A diff file is composed of one or more code segments that are used as the characteristic fingerprint of the corresponding vulnerable code. In a diff file, there are code statements identified by special notations. Statements beginning with "+" are statements added by the patch, and statements beginning with "-" are statements deleted by the patch. Finally, the vulnerable functions, the vulnerable code segments, and the statements added and deleted by each vulnerability patch for vulnerable source codes are saved in a local file library for subsequent fingerprint generation and clone detection of vulnerable codes.

3.2. Preprocessing. The vulnerable code segments in a patch cannot completely represent the context of the vulnerable code. To better obtain the semantic information of vulnerable code, the vulnerable functions are converted into code flow graphs, and vulnerability patch code control statements with contextual information are notated for subsequent code clone detection. The Code2flow tool can convert vulnerable functions into flow graphs to determine and notate the control statements corresponding to patch codes [37]. Figure 1 shows an example of a simple vulnerable function and its generated flow diagram.

It is necessary to preprocess the source code before generating fingerprints. The first step of preprocessing the source code is to normalize the vulnerable source code by deleting annotations, spaces, tabs, and line breaks and converting all characters into lowercase letters to eliminate the influence of factors unrelated to syntax on the detection results. The steps of abstract replacement in the source code are as follows.

Step 1. Formal parameter replacement. Replace the formal parameters of functions in the code with FPARAM symbols.

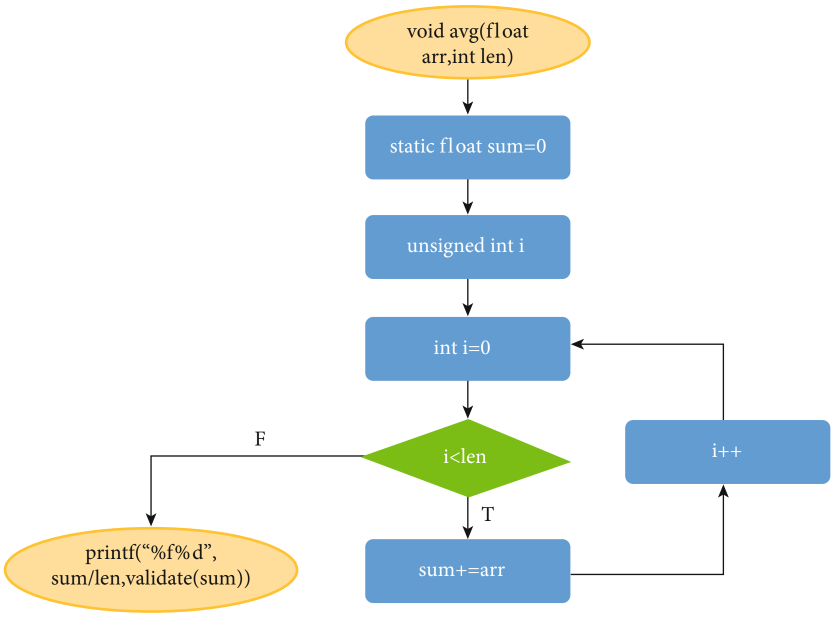

void avg(float arr,int len)
static float sum=0
unsigned int i
int i=0
i<len
F
T
i++
printf("%f%d", sum/len,validate(sum))
sum+=arr

Figure 1: An instance of Code2flow for vulnerable function.

Step 2. Local variable replacement. Replace local variables in the code with LVAR symbols.

Step 3. Data type replacement. Replace the data types in the code with DTYPE symbols. These data types include C-language data types and custom data types. Modifiers such as unsigned will not be replaced because they have a significant impact on some vulnerabilities, such as integer overflow.

Step 4. Function replacement. Replace the function calls in the code with FUNCCALL symbols. Function calls are an important source of vulnerability.

Figure 2 shows the transformation of a vulnerable function before and after preprocessing. Figure 2(a) shows the vulnerable function code before preprocessing, and Figure 2(b) shows the vulnerable function code after preprocessing.

3.3. Granularity Selection. For Type 1 and Type 2 code clones, clone detection is performed on the preprocessed vulnerable code with function-level granularity, which takes less time than detection with line-of-code-level granularity. However, in the face of Type 3 and Type 4 code clones, it is difficult for detection with function-level granularity to succeed because it ignores the possible methods of internal modification of vulnerable functions, such as modification of the code statement sequence or the insertion of redundant code. Thus, to detect multiple types of clones, the proposed method combines line-level granularity and function-level granularity for detection. First, fingerprints of vulnerable codes are generated to detect Type 1 and Type 2 code clones with the vulnerable functions as the units for detection, and then, fingerprints are generated to detect Type 3 and Type 4 code clones with code lines as the units. If only the vulnerable code segments themselves are used to generate the fingerprints, contextual information may be omitted. Thus, the control statements corresponding to vulnerability patch codes are also selected for fingerprint generation. Finally, the added and deleted statements in each patch file are also used to generate fingerprints for subsequent patch verification.

Figure 3 shows the process of generating fingerprints of vulnerable code in detail. The preprocessed vulnerable source code is divided into a vulnerable function code set and a vulnerable line-level code set for the vulnerable code segments, the vulnerability patch codes, and the control statements corresponding to the vulnerability patch codes. Fingerprints of vulnerable functions are generated at the function level and stored in the fingerprint database. Then, fingerprints of the vulnerable code segments, vulnerability patch codes, and control statements corresponding to vulnerability patch codes in the vulnerable line-level code set are generated at the line-of-code level of granularity. The line-level code fingerprints are associated with the corresponding vulnerable function fingerprints to complete the construction of the whole fingerprint database.

3.4. Fingerprint Generation. In the process of detecting Type 1 and Type 2 code clones, a triple $t = (l, h, f)$ represents a vulnerable function fingerprint. Here, l represents the length of the vulnerable function, h is the Common Vulnerabilities and Exposures (CVE) identification number of the vulnerability corresponding to the vulnerable function, and f denotes the hash value of the vulnerable function fingerprint. Vulnerable function fingerprints of the same length are stored as one data set. Thus, CPVDetector can locate vulnerable function fingerprints based on the vulnerable function length and can quickly search for any associated CVE number that may exist in the target code to be detected. Table 1 presents the vulnerable function fingerprints of the vulnerabilities numbered CVE-2017-13012 and CVE-2015-1308, with a vulnerable function length of 143.

Step 1: Formal parameter replacement	DumpStyleGeneaology(nsIFrame* fparam, const char* fparam)
	nsFrame::ListTag(fparam, fparam);
	nsStyleContext* sc = aFrame->GetStyleContext();
	printf("%p", fparam);
	psc = sc->GetParent();
	sc = psc;
	printf("%p ", fparam);

Step 2: Local variable replacement.	DumpStyleGeneaology(nsIFrame* fparam, const char* fparam)
	nsFrame::ListTag(fparam, fparam);
	nsStyleContext* lvar = aFrame->GetStyleContext();
	printf("%p ", fparam);
	lvar = sc->GetParent();
	lvar = lvar;
	printf("%p ", fparam);

Step 3: Data type replacement.	DumpStyleGeneaology(dtype* fparam, const dtype* fparam)
	nsFrame::ListTag(fparam, fparam);
	dtype* lvar = aFrame->GetStyleContext();
	printf("%p ", fparam);
	lvar = sc->GetParent();
	lvar = lvar;
	printf("%p ", fparam);

Step 4: Function replacement.	DumpStyleGeneaology(dtype* fparam, const dtype* fparam)
	nsFrame::funccall(fparam, fparam);
	dtype* lvar = aFrame->funccall();
	funccall("%p ", fparam);
	lvar = sc->funccall();
	lvar = lvar;
	funccall("%p ", fparam);

Figure 2: Example of line-by-line abstract replacement.

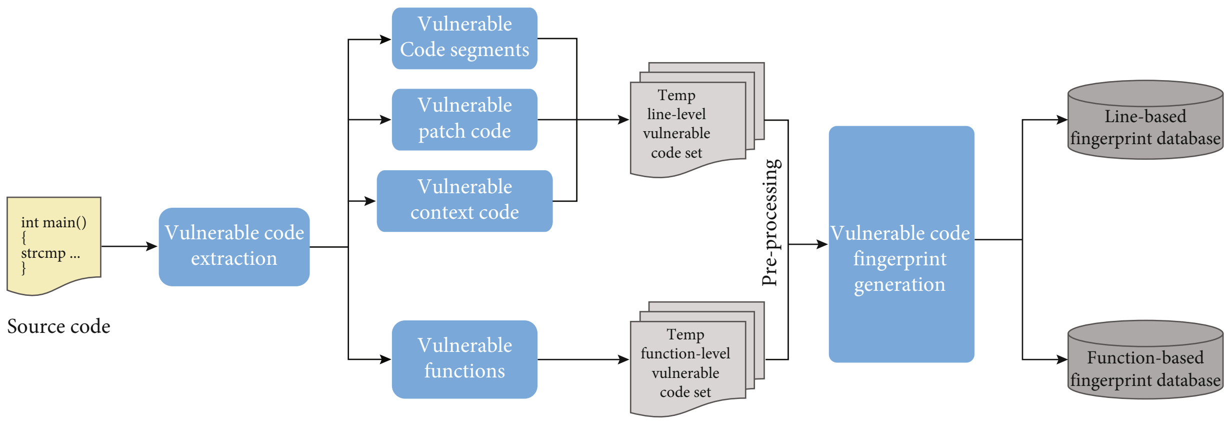

int main()
{
strcmp ...
}
Source code
Vulnerable code extraction
Vulnerable Code segments
Vulnerable patch code
Vulnerable context code
Vulnerable functions
Temp line-level vulnerable code set
Temp function-level vulnerable code set
Pre-processing
Vulnerable code fingerprint generation
Line-based fingerprint database
Function-based fingerprint database

Figure 3: Construction of the fingerprint database.

The vulnerable line-level code fingerprints for Type 3 and Type 4 code clones can be divided into three types: (i) fingerprints of vulnerable code segments, generated by rows; (ii) fingerprints of the control statements corresponding to vulnerability patch codes, used to further verify the context of cloned code when detecting code clones; and (iii) fingerprints of vulnerability patch codes, including fingerprints of patch-deleted statements marked with "-" and fingerprints of patch-added statements marked with "+." The specific operations are as follows. Fingerprints of the vulnerable code segments in the vulnerable line-level code set are generated, followed by the fingerprints of the corresponding vulnerability patch code control statements and vulnerability patch codes at line-level granularity, and the above three

Table 1: Examples of vulnerable function fingerprints.

Vulnerable function length	CVE number of the vulnerability	Vulnerable function fingerprint
143	CVE-2017-13012	08345519ec358d8af82812efab5051f6
	CVE-2015-1308	78675ef39f395a7d95076b3354f0e48e

types of fingerprints are stored in a table space as the fingerprints of the vulnerable line-level codes. Finally, each vulnerable line-level code fingerprint is linked to the corresponding vulnerable function fingerprint to allow the corresponding vulnerability CVE number to be output in the results of code clone detection. Table 2 presents the preprocessed vulnerable code and vulnerable line-level code fingerprints of the vulnerable code segments, patch code control statements, and vulnerability patch code for CVE-2017-13012.

4. Code Clone Detection

CPVDetector first eliminates the influence of operations such as renaming in the target code on the clone detection results. Second, each function in the target code is converted into a fingerprint to detect Type 1 and Type 2 code clones. Once any matching vulnerable function fingerprint is discovered, detection is terminated, and the CVE number corresponding to the vulnerable function fingerprint will be output to end the detection process. If no corresponding vulnerable function fingerprint is discovered, the target code should be processed with lines of code as the units. Then, Type 3 and Type 4 code clones should be detected in the fingerprint database of vulnerable line-level codes, and the context of the vulnerable codes and any corresponding patches should be further verified in the process of detecting target line-level code clones. Finally, the detection results are output. The process of detecting code clones with CPVDetector is shown in Figure 4.

4.1. Function-Level Vulnerable Code Clone Detection. Vulnerable function fingerprints are employed to detect Type 1 and Type 2 code clones. In the detection process for the target code, the length of the vulnerable function is first used as an index to search for any vulnerable function of the same length. If any vulnerable function fingerprint with the same length is discovered, hash lookup will be used to match hash values with the given length, which may help reduce the search space and improve the detection efficiency. As a result, the CVE number corresponding to the vulnerable function fingerprint will be output. Fingerprints of vulnerable functions that may have been cloned can be quickly located with an average time complexity of $O(1)$ and a worst-case time complexity of $O(n)$.

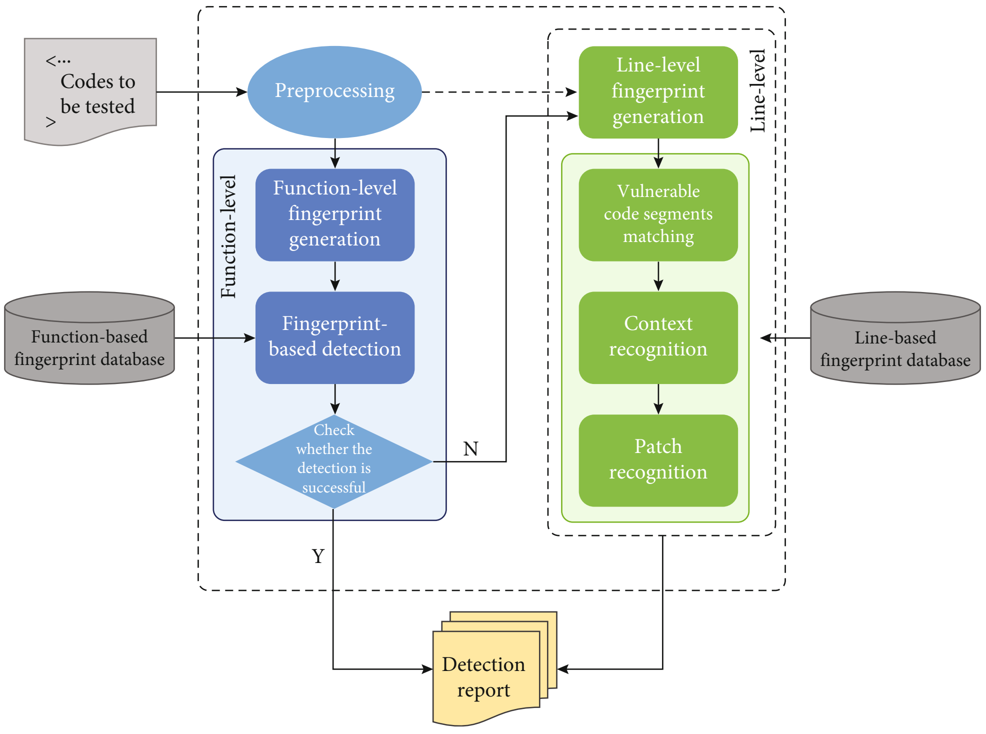
4.2. Line-Level Vulnerable Code Clone Detection. Line-level clone detection is mainly used for the detection of Type 3 and Type 4 code clones. Detection based on the fingerprint of the vulnerable code segment is performed first. In general, there will be a vulnerability in target code that includes an entire vulnerable code segment. In other words, fingerprint detection for vulnerable code segments can be regarded as a problem of finding whether a given subsequence is present. Therefore, we introduce a greedy algorithm to handle this problem. The fingerprints in the vulnerable line-level code fingerprint database are used as the input to match against the target code. If a vulnerable code segment associated with a vulnerability is found, it can be judged that the target code may contain this vulnerability. Algorithm 1 describes the detection algorithm for vulnerable line-level code clones.

In Algorithm 1, T represents the set of line-level fingerprints of the target code, and F represents the set of line-level fingerprints of a given vulnerable code segment. t and f represent the MD5 fingerprints at the line-of-code level in T and F, respectively. The symbol $|\bullet|$ denotes the size of the set $\bullet$. If the size of T is smaller than the size of F, then the vulnerable code cannot appear as a subsequence in the destination code, which means that there is no successful match. After this conditional judgment, the corresponding vulnerable code fingerprints are matched in the destination code, and the number of matches is recorded by the counter j. When the value of the counter j is equal to the length of the vulnerable code fingerprint, this indicates that the complete sequence of the vulnerable code exists in the destination code, which means that the match is successful. As is clearly seen from Algorithm 1, the complexity of Algorithm 1 is $O(|T| \times |F|)$ in the worst case, depending on the number of lines in the target code and the number of fingerprints of the vulnerable code segment.

4.3. Vulnerability Context and Patch Validation. If the target code is matched only with the fingerprints of the vulnerable code segment, the following two problems may affect the detection result. (i) The context of the vulnerable code is difficult to completely express. When the context of the vulnerable code has changed, the detection method may fail to recognize it. (ii) Even if the vulnerable code segment in the target code has been patched, the detection method may still indicate that there is a vulnerability in the target code. To address these two problems, the proposed method verifies the vulnerability context of the target code. Because a vulnerability patch is designed to fix a specific vulnerability and the patch code can only work if it acts on the corresponding vulnerable code, the contextual relationship is the same for the patch code and the corresponding vulnerable code. Therefore, before patch verification, the contextual relationship is first verified to ensure that the contextual relationship of the vulnerable code has not changed and to guarantee that the patch code can be successfully applied to the corresponding vulnerable code. If the target code successfully passes the vulnerable code segment detection process, the previously introduced line-level vulnerable code clone detection algorithm will be used to verify the vulnerability context, and the target code and the fingerprints of corresponding

Table 2: Examples of vulnerable line-level code fingerprints.

Type	Preprocessed vulnerable code	Vulnerable line-level code fingerprint
Vulnerable code segments	*FPARAM+=8;*	*62e857215c2c8b10a1ebe99046b1b463*
	FUNCCALL((FPARAM, "\n\t"));	*04a27a8b584e00db088f18a16ddc1ac6*
	FPARAM=(const struct FPARAM ∗)LVAR;	*c3250b774d4bd8bbeea848ae8e091777*
	FPARAM=FPARAM;	*4830812a7240fa89419da5ef1e440566*
	LVAR=FPARAM->FPARAM;	*01689c59d66b453ae92f59560a0a430f*
	FUNCCALL(FPARAM,LVAR,FUNCCALL(&FPARAM));	*d1f3dbd2ab6d7e6199391bb86d1d343a*
	FPARAM=FPARAM;	*57e012af713ea16d25ed7bd54a216887*
Vulnerability patch code	*FUNCCALL(&FPARAM);*	+90b650dba71b052ca2a85608a2447c1c
	FUNCCALL(&FPARAM->FPARAM);	-e58a03ee48a74c4d50af5bee1ba18d08
Vulnerability context code	*if(FPARAM>=1&&FUNCCALL(FPARAM))*	*8fddbc554a5ad97e8805a956a8ab5968*



<...
Codes to
be tested
>
Preprocessing
Function-level
Function-level fingerprint generation
Fingerprint-based detection
Check whether the detection is successful
N
Y
Function-based fingerprint database
Line-level
Line-level fingerprint generation
Vulnerable code segments matching
Context recognition
Patch recognition
Line-based fingerprint database
Detection report

Figure 4: Framework of code clone detection in CPVDetector.

patch code control statements will jointly serve as the input to the detection algorithm.

To lower the false positive rate caused by the neglect of patch information, a method proposed in this paper uses the target code passing the context verification as the input for the vulnerable patch validation. This method is different from the previous detection method. The package of vulnerability patch codes can be divided into patch-added statements and patch-deleted statements. Based on this premise, for vulnerability patch validation, the fingerprints of statements identified by "-" as patch deletions in the patch code will firstly be detected in the target code. If these deleted patch code fingerprints are not detected, the fingerprints of statements identified by "+" as patch additions will be detected in the target code. If no fingerprints of the deleted statements are detected and the fingerprints of the added statements are detected in the target code, it is identified that the vulnerabilities in the code have been patched, and the false positives caused by patching are thus eliminated.

5. Experiment

To evaluate the effectiveness of the proposed method, it is compared with VUDDY, ReDeBug, and Deckard in this section. In the experiment, common C/C++ open-source software projects obtained from GitHub, such as the Linux kernel and FFmpeg, and their vulnerability information and patch files were obtained from the Common Weakness Enumeration (CWE) website to establish a complete vulnerability code fingerprint database. Moreover, many test cases for different vulnerabilities have been posted in the Software Assurance Reference Dataset (SARD) [38], including vulnerable

```
Input: $T$, $F$
Output: $r$
Initialize: $r \leftarrow$ False
1. if $|T| < |F|$ then
2.     return $r$
3. end if
4. for each $t$ in $T$ do
5.   $j \leftarrow 0$
6.   for each $f$ in $F$ do
7.     if $t = f$ then
8.       goto 14
9.     end if
10.    $j \leftarrow j+1$
11.    if $j = |F|$ then
12.      return $r$
13.    end if
14.  end for
15. end for
16. $r \leftarrow$ True
17. return $r$
```

Algorithm 1: Line-level vulnerable code clone detection algorithm.

Table 3: Evaluation metrics.

Evaluate indicator	Formula
Precision	P = TP/(TP + FP)
Accuracy	A = (TP + TN)/(TP + FP + TN + FN)
False positive rate	FPR = FP/(FP + TN)
False negative rate	FNR = FN/(FN + TP)
F-measure	F-Measure = (2 ∗ P ∗ (1-FNR))/(P + (1-FNR))

Table 4: Experimental results of different methods on the test set.

Method	P (%)	A (%)	FPR (%)	FNR (%)	F-measure (%)
CPVDetector	96.47	92.94	2.35	10.09	93.07
VUDDY	94.11	75.29	7.06	56.47	59.52
ReDeBug	91.76	67.05	8.23	57.64	57.96
Deckard	57.65	50.59	47.05	55.29	50.36

cases, nonvulnerable cases, and patched cases, to build a set of test cases for experimentation.

5.1. Evaluation Indicators. To verify the effectiveness of each model, the precision (P), accuracy (A), false positive rate (FPR), false negative rate (FNR), and F-measure were used as the performance indicators in this study. In the expressions for these indicators, TP represents the number of samples correctly detected as vulnerabilities, FP is the number of samples incorrectly detected as vulnerabilities, TN represents the number of samples correctly detected as nonvulnerabilities, and FN is the number of samples incorrectly detected as nonvulnerabilities. The specific calculation formulas for the 5 abovementioned indicators are shown in Table 3.

5.2. Accuracy Evaluation. In the comparative test, the size of the ReDeBug sliding window was set to 4, and the length of the extracted code segment was set to 10. In Deckard, the minimal number of tokens required for clones was 30, the size of the sliding window was set to 2, and the similarity value was set to 0.95. The computer used to run the experiment was configured with an AMD Ryzen 5 3600 CPU, an Nvidia RTX 2060S 8 GB GPU, 16 GB of memory, and a 500 GB SSD.

The experimental results are shown in Table 4. Compared with the results of the other three methods, the F-measure of the proposed method is increased by at least 33%. Meanwhile, the proposed CPVDetector has higher precision and accuracy, indicating that more vulnerabilities can be detected without incurring more false positives. CPVDetector preserves the context information of vulnerable code, generates more accurate code fingerprints, and thus achieves a higher accuracy of detection. Furthermore, it is low for the FNR of CPVDetector, reaching approximately 10%, i.e., a quarter of that of the other methods, which suggests that false positives can be reduced by means of context and patch validation. VUDDY and ReDeBug use code fingerprinting to characterize code, but due to the choice of granularity, incomplete analysis of semantic information, and incomplete consideration of specific situations, they result in high FPRs and an inability to effectively deal with different types of code clones. Meanwhile, Deckard is an abstract-syntax-tree-based code clone detection method, and syntax tree generation and subtree finding are relatively inefficient processes. This results in low overall efficiency of the method, and the fact that detection is performed only on the basis of structural similarities can also lead to false positives due to the neglect of changes in internal details, which is also the reason for the relatively low detection precision and accuracy. An experimental analysis of different types of code cloning and the corresponding detection efficiency is as follow.

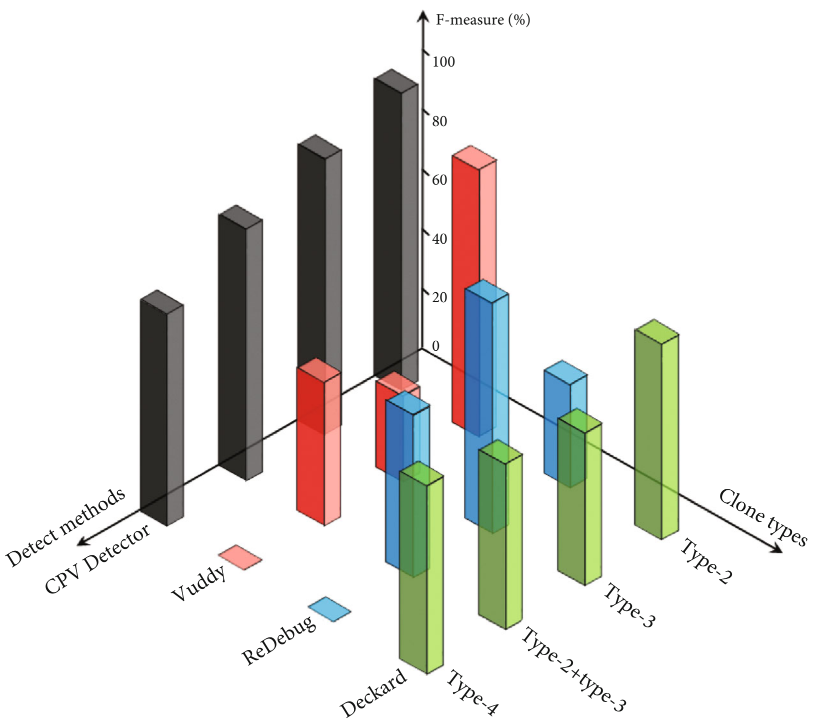


Figure 5: F-measure comparisons for different clone types and detection methods.

To study the detection effect of the proposed method for different types of code clones, the cases were divided into 4 test sets, i.e., Type 2 code clones, Type 3 code clones, mixed Type 2 and Type 3 code clones, and Type 4 code clones. CPVDetector, VUDDY, ReDeBug, and Deckard were all tested on each of these 4 test sets individually.

Figure 5 depicts the experimental results of the 4 methods for detecting different types of code clones. As shown in Figure 5, the F-measure of the method presented herein was increased by at least 7%, 19%, and 17% compared with the other three methods in terms of detecting Type 2, Type 3, and Type 4 code clones, respectively, and can effectively detect different types of code clones. VUDDY has good performance in detecting Type 2 code clones, but for Type 3, its F-measure was 13.6%, and it also could not detect Type 4 code clones. The F-measure of ReDeBug was only 27.3% in detecting Type 2 code clones, and it could not detect Type 4 code clones either. Although Deckard works for all different types of code clones, its F-measure for the detection of Type 2 code clones was only 52%. In summary, the experimental results of the other three types of detection methods were not ideal; accordingly, the main problems of VUDDY, ReDeBug, and Deckard are analyzed as follows. Because function-level granularity is used for detection in VUDDY, it cannot effectively detect clones in cases of insertion of redundant code or deletion of irrelevant code. Therefore, VUDDY does not have an ideal effect in detecting Type 3 and Type 4 code clones. For ReDeBug, it is difficult to detect cloned codes whose data types have been changed or whose variables and functions have been renamed, which leads to a serious problem of false negatives in the detection of Type 2 code clones. Deckard converts code into an abstract syntax tree and detects code clones by searching for similar subtrees; consequently, target code with subtrees similar to existing vulnerabilities and without actual code clones will be falsely determined as vulnerable one because of the lack of patch validation.

Table 5: Time overhead comparisons.

LoC	CPVDetector	VUDDY	ReDeBug	Deckard
1k	0.62	0.44	11.6	1.12
10k	1.45	0.81	27.6	3.63
50k	6.47	4.86	38.5	14.27
100k	17.63	10.34	42.92	34.67
200k	38.44	24.81	103.43	185.69
300k	65.12	47.43	195.24	332.42

For experiments to evaluate the detection efficiency of the proposed method, the test cases were divided into sets of different sizes, ranging from 1k LoC to 300k LoC. Table 4 lists the run times of the different methods on the test case sets of different sizes. As seen in Table 5, the proposed method shows significantly higher efficiency than ReDeBug and Deckard. In these experiments, Deckard required considerably more time as the data scale increased because this detection method is based on abstract syntax trees. VUDDY consumed less time than the other methods because it performs detection at the function level granularity; however, as a result, it cannot effectively detect Type 3 and Type 4 code clones, so a high FPR is generated. Compared with VUDDY, the proposed method

```
if (ndo->ndo_vflag >= 1 &&
    ICMP_ERRTYPE(dp->icmp_type)){
  bp += 8;
  ND_PRINT((ndo, "\n\t"));
  ip = (const struct ip *)bp;
  snapend_save = ndo->ndo_snapend;
  ip_print(ndo, bp, EXTRACT_16BITS
        (&ip->ip_len));
  ndo->ndo_snapend = snapend_save;
}
```

(a) Vulnerability source code of CVE-2017–13012

```
if (ndo->ndo_vflag >= 1 &&
    ICMP_ERRTYPE(dp->icmp_type)){
  bp += 8;
  ND_PRINT((ndo, "\n\t"));
  ip = (const struct ip*)bp;
  snapend_save = ndo->ndo_snapend;
  ND_TCHECK_16BITS(&ip->ip_len);
  ip_print(ndo, bp, EXTRACT_16BITS
    (&ip->ip_len));
  ndo->ndo_snapend = snapend_save;
}
```

(b) Patched code of CVE-2017–13012

```
if (ndo->ndo_vflag >= 1 &&
    ICMP_ERRTYPE(dp->icmp_type)){
  bp += 8;
  ND_PRINT((ndo, "\n\t"));
  ip = (const struct ip *)bp;
  ndo->ndo_snaplen = ndo->
  ndo_snapend - bp;
  snapend_save = ndo->ndo_snapend;
  ip_print(ndo, bp, EXTRACT_16BITS
        (&ip->ip_len));
  ndo->ndo_snapend = snapend_save;
}
```

(c) Test case of CVE-2017–13012

Figure 6: Three versions of the vulnerability CVE-2017-13012.

shows improved efficiency in detecting different types of code clones at the expense of a small increase in run time. In addition, the time overhead of CPVDetector exhibits roughly linear growth as the size of the test case set increases, indicating that it can be feasibly applied in large-scale code detection scenarios.

5.3. Case Analysis of Vulnerable Code Clone Detection. In this section, three specific cases are analyzed for vulnerable code clone detection. Taken the vulnerability CVE-2017-13012 as an example, the proposed method could successfully detect the vulnerability, whereas VUDDY, ReDeBug, and Deckard all failed. CVE-2017-13012, a high-risk vulnerability with a Common Vulnerability Scoring System (CVSS) risk score of 9.8, is generated when the ICMP parser in tcpdump suffers from buffer overreading. Figure 6 shows three versions of the vulnerability CVE-2017-13012. Figure 6(a) presents the original source code that led to this vulnerability. The calls of *ip_print* and *EXTRACT_16BITS (&ip->ip len)* lead to the occurrence of this vulnerability. Therefore, it is meaningful to check the value of *ip->ip_len* before using it. Figure 6(b) presents the patched code in which this vulnerability is corrected, with the addition of the call to the statement of *ND_TCHECK 16BITS(&ip->ip_len)*. A test case of the constructed test set is shown in Figure 6(c). It can be seen that the callee function is not present, which suggests that this code has not been repaired by the vulnerability patch and that consequently, the vulnerability may still exist. Moreover, the statement *ndo->ndo_snaplen=ndo->ndo_snapend-bp* has been added in Figure 6(c); this statement does not affect the vulnerable code because it cannot prevent buffer overreading caused by *ip_print* and *EXTRACT_16BITS (&ip->ip len)*.

Second, the vulnerability CVE-2018-18314 is a buffer overflow vulnerability that occurs via a crafted regular expression that triggers invalid write operations. Its patch fixes the vulnerability by adding multiple sets of conditional validations to the regcomp.c file. A sample code for a version of this vulnerability exists in the test set. Although this sample also adds conditional judgments to further validate the input regular expressions, there are individual conditional validation statements that are not added in the appropriate places, and their contextual relationships are broken, causing the patch code to fail to work. For example, an *if (UCHARAT(RExC_parse)! = ')')* statement needs to be added in the *S_handle_regex_sets* function, and *RExC_parse* needs to be updated with *switch* judgment rather than *if* statement. This vulnerability has a CVSS score of 9.8; it affects products such as MySQL, Oracle, and Solaris and requires patching to be completed as soon as possible.

Finally, the vulnerability CVE-2020-25643 was detected in the Linux kernel. With a CVSS value of 7.5, it is also a high-risk vulnerability. This vulnerability is generated when incorrect input verification in the *ppp_cp_parse_cr* function of the HDLC_PPP module of the Linux kernel causes memory corruption and read overflow, and it can lead to system denial of service or direct breakdown. This vulnerability was not patched in Fedora after it was repaired in the Linux kernel, and consequently, the risk still exists. In addition, there are vulnerabilities such as CVE-2016-9376, CVE-2018-13014, CVE-2018-10937, and CVE-2019-12981, which are not detected by the other three code clone detection methods, while the method proposed in this paper can effectively detect these vulnerabilities from source code by detecting them at different granularities.

6. Conclusion

This paper presents a vulnerable code clone detection tool, CPVDetector, based on code fingerprints with context and patch validation. Vulnerable code can be located via the corresponding diff file and is preprocessed to eliminate the impact of extraneous elements of the code and the renaming operation on detection. First, code clone detection is carried out at the function level for the target code. Then, context and patch validation of the vulnerable code is implemented in line-level code clone detection, which can obviously reduce the false positive rate. CPVDetector can effectively detect vulnerable code clones of Type 1, Type 2, Type 3, and Type 4. Experimental results show that the proposed method achieves an accuracy of 92.94% and a precision of 96.47%, resulting in an improvement in the F-measure by at least 7% compared with other methods, and can discover vulnerable code clones that other methods cannot detect. In terms of time overhead, CPVDetector is close to VUDDY, but CPVDetector outperforms VUDDY in detection. Our future work is as follows. First, some abstraction methods can be combined with vulnerable code extraction to retain more semantic information in order to verify the contextual relationships. Second, the vulnerable code fingerprint database can be further complemented and enriched by considering vulnerable codes written in different languages to broaden the application scope of the method. Finally, machine learning or deep learning can be considered to improve the effectiveness of the detection approach.

Data Availability

No datasets available.

Conflicts of Interest

The authors declare that they have no conflicts of interest.

Acknowledgments

This work was partly supported by the National Nature Science Foundation of China (Grant No. 52072293), the Scientific Research Program Funded by the Shaanxi Provincial Education Department (Program No. 19JC021), the Science and Technology Project Funded by Xi'an City in Shaanxi Province of China (Program No. 2020KJRC0039), and the Science and Technology Project Funded by Weiyang District in Xi'an City, Shaanxi Province, China (Program No. 201922).

References

[1] W. Scacchi, "Understanding open source software evolution 181," *Software Evolution and Feedback: Theory and Practice*, vol. 181-205, 2006.

[2] X. Tan, M. Zhou, and B. Fitzgerald, "Scaling open source communities: an empirical study of the Linux kernel," in *2020IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, pp. 1222–1234, Seoul, Korea (South), 2020.

[3] "GitHub," https://github.com.

[4] N. Nurmuliani, D. Zowghi, and S. Powell, "Analysis of requirements volatility during software development life cycle," in *2004 Australian Software Engineering Conference. Proceedings*, pp. 28–37, Melbourne, VIC, Australia, 2004.

[5] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on Software Engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[6] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queen's School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[7] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: a systematic review," *Information and Software Technology*, vol. 55, no. 7, pp. 1165–1199, 2013.

[8] G. Shobha, A. Rana, V. Kansal, and S. Tanwar, "Code clone detection—a systematic review," *Emerging Technologies in Data Mining and Information Security*, vol. 1300, 2021.

[9] Y. Wu, D. Zou, S. Dou et al., "SCDetector: software functional clone detection based on semantic tokens analysis," in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pp. 821–833, Australia, 2020.

[10] I. Keivanloo and J. Rilling, *Source Code Clone Search [M]// Code Clone Analysis*, Springer, Singapore, 2021.

[11] D. Y. Lee, U. Ko, I. Aitkazin, S. Park, H. S. Tak, and H. G. Cho, "A fast detecting method for clone functions using global alignment of token sequences," in *Proceedings of the 2020 12th International Conference on Machine Learning and Computing*, pp. 17–22, Shenzhen, China, 2020.

[12] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "Deckard: scalable and accurate tree-based detection of code clones," in *29th International Conference on Software Engineering (ICSE'07)*, pp. 96–105, Minneapolis, MN, USA, 2007.

[13] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: finding unpatched code clones in entire OS distributions," in *2012 IEEE symposium on security and privacy*, pp. 48–62, San Francisco, CA, USA, 2012.

[14] P. M. Caldeira, K. Sakamoto, H. Washizaki, Y. Fukazawa, and T. Shimada, "Improving syntactical clone detection methods through the use of an intermediate representation," in *2020 IEEE 14th international workshop on software clones (IWSC)*, pp. 8–14, London, ON, Canada, 2020.

[15] S. Kim, S. Woo, H. Lee, and H. Oh, "Vuddy: a scalable approach for vulnerable code clone discovery," in *2017 IEEE symposium on security and privacy (SP)*, pp. 595–614, San Jose, CA, USA, 2017.

[16] W. Jiang, B. Wu, Z. Jiang, and S. Yang, "Cloning vulnerability detection in driver layer of IoT devices," in *International Conference on Information and Communications Security*, pp. 89–104, Springer, Cham, 2020.

[17] F. Cheirdari and G. Karabatis, "Analyzing false positive source code vulnerabilities using static analysis tools," in *2018 IEEE International Conference on Big Data*, pp. 4782–4788, Seattle, WA, USA, 2018.

[18] S. Kim and H. Lee, "Software systems at risk: an empirical study of cloned vulnerabilities in practice," *Computers & Security*, vol. 77, pp. 720–736, 2018.

[19] X. S. Cybersecurity, "Dynamics: a foundation for the science of cybersecurity," in *Proactive and Dynamic Network Defense*, pp. 1–31, Springer, Cham, 2019.

[20] G. Lin, J. Zhang, W. Luo et al., "Cross-project transfer representation learning for vulnerable function discovery," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 7, pp. 3289–3297, 2018.

[21] A. S. Bin-Habtoor and M. A. Zaher, "A survey on plagiarism detection systems," *International Journal of Computer Theory and Engineering*, vol. 4, no. 2, pp. 185–188, 2012.

[22] E. L. Jones, "Metrics based plagarism monitoring," *Journal of Computing Sciences in Colleges*, vol. 16, no. 4, pp. 253–261, 2001.

[23] D. Budgen and P. Brereton, "Performing systematic literature reviews in software engineering," in *Proceedings of the 28th*

international conference on Software engineering, pp. 1051-1052, Shanghai, China, 2006.

[24] C. Ragkhitwetsagul and J. Krinke, "Using compilation/decompilation to enhance clone detection," in *2017 IEEE 11th International Workshop on Software Clones (IWSC)*, vol. 1-7, Klagenfurt, Austria, 2017.

[25] C. K. Roy and J. R. Cordy, "NICAD: accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *2008 16th IEEE international conference on program comprehension*, pp. 172–181, Amsterdam, Netherlands, 2008.

[26] S. Jadon, "Code clones detection using machine learning technique: Support vector machine," in *2016 International Conference on Computing, Communication and Automation (ICCCA)*, p. 399, Greater Noida, India, 2016.

[27] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[28] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: scaling code clone detection to big-code," in *Proceedings of the 38th International Conference on Software Engineering*, pp. 1157–1168, Austin, Texas, 2016.

[29] M. A. Nishi and K. Damevski, "Scalable code clone detection and search based on adaptive prefix filtering," *Journal of Systems and Software*, vol. 137, no. MAR., pp. 130–142, 2018.

[30] P. Wang, J. Svajlenko, Y. Wu, Y. Xu, and C. K. Roy, "CCAligner: a token based large-gap clone detector," in *Proceedings of the 40th International Conference on Software Engineering*, pp. 1066–1077, Gothenburg, Sweden, 2018.

[31] Y. Yang, Z. Ren, X. Chen, and H. Jiang, "Structural function based code clone detection using a new hybrid technique," in *2018 IEEE 42nd annual computer software and applications conference (COMPSAC)*, pp. 286–291, Tokyo, Japan, 2018.

[32] J. Crussell, C. Gibler, and H. Chen, "Andarwin: scalable detection of android application clones based on semantics," *IEEE Transactions on Mobile Computing*, vol. 14, no. 10, pp. 2007–2019, 2015.

[33] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt, "Practical and optimal LSH for angular distance," 2015, https://arxiv.org/abs/1509.02897.

[34] J. Svajlenko and C. K. Roy, "Fast and flexible large-scale clone detection with CloneWorks," in *ICSE (Companion Volume)*, pp. 27–30, Buenos Aires, Argentina, 2017.

[35] G. Singh, "To enhance the code clone detection algorithm by using hybrid approach for detection of code clones," in *2017 International Conference on Intelligent Computing and Control Systems (ICICCS)*, pp. 192–198, Madurai, India, 2017.

[36] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "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, Los Angeles, California, USA, 2016.

[37] F. Ji, "A method of hierarchical reconfiguration of flow chart reversing from C source code," *Software Engineering and Applications*, vol. 7, no. 3, pp. 168–176, 2018.

[38] "SARD," https://samate.nist.gov/SARD/.