IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 25, 2021, accepted February 22, 2021, date of publication March 15, 2021, date of current version April 2, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3065872

A Survey of Software Clone Detection From Security Perspective

HAIBO ZHANG[1] AND KOUICHI SAKURAI[2], (Member, IEEE)

[1]Department of Informatics, Graduate School of Information Science and Electrical Engineering, Kyushu University, Fukuoka 819-0395, Japan
[2]Department of Informatics, Faculty of Information Science and Electrical Engineering, Kyushu University, Fukuoka 819-0395, Japan

Corresponding author: Haibo Zhang (zhang.haibo@inf.kyushu-u.ac.jp)

This work was supported in part by the Collaboration Hubs for International Program (CHIRP) of Strategic International Collaborative Research Program (SICORP) through the Japan Science and Technology Agency (JST).

ABSTRACT For software engineering, if two code fragments are closely similar with minor modifications or even identical due to a copy-paste behavior, that is called software/code clone. Code clones can cause trouble in software maintenance and debugging process because identifying all copied compromised code fragments in other locations is time-consuming. Researchers have been working on code clone detection issues for a long time, and the discussion mainly focuses on software engineering management and system maintenance. Another considerable issue is that code cloning provides an easy way to attackers for malicious code injection. A thorough survey work of code clone identification/detection from the security perspective is indispensable for providing a comprehensive review of existing related works and proposing future potential research directions. This paper can satisfy above requirements. We review and introduce existing security-related works following three different classifications and various comparison criteria. We then discuss three further research directions, (i) deep learning-based code clone vulnerability detection, (ii) vulnerable code clone detection for 5G-Internet of Things devices, and (iii) real-time detection methods for more efficiently detecting clone attacks. These methods are more advanced and adaptive to technological development than current technologies, and still have enough research space for future studies.

INDEX TERMS Code clone, security analysis, software clone, vulnerability detection.

I. INTRODUCTION

In the field of software development, programmers prefer to copy and paste a piece of source code directly from another source code fragment, even if there are minor modifications, so that they look similar or even identical. This is called software/code cloning [1], [2], some researchers also call it code duplication [3]–[5]. Many reasons exist for code cloning; the main reason is that code clones can help programmers to finish their tasks more quickly. Programming and maintenance issues occur because of this type of behavior. For instance, if a bug is found in a cloned code fragment of a software system, the programmer has to detect this bug everywhere and fix it, which increases software maintenance difficulties [1].

Furthermore, in terms of software system security, code clones could lead to vulnerability propagation if a vulnerable code fragment is cloned [6]. Even though software programmers are trying to write secure source code and minimize vulnerabilities in the source code when developing their systems [7], code clone behavior inevitably occurs during the software programming process and propagates system vulnerabilities [8], [9].

The associate editor coordinating the review of this manuscript and approving it for publication was Mansoor Ahmed.

Code clone vulnerability detection has been studied intensively. Jiang *et al.* [10] presented a scalable and accurate approach for detecting code clones on the basis of identifying similar subtrees. Yamaguchi *et al.* [11] proposed a method called Chucky to statically taint source code, which can identify anomalous or missing conditions linked to security-critical objects. Farhadi *et al.* [12] presented a malicious code clone detection technique for binary code based on token normalization levels. Some researchers have also applied more advanced and efficient technologies to improve the efficiency of vulnerability detection systems. Li *et al.* [13] proposed a deep learning-based system for source code vulnerability detection called VulDeePecker. For applications based on code clone detection, Gao *et al.* [14] applied an approach based on binary vulnerability search for cross-platform Internet of Things (IoT) devices. Hum *et al.* [15] proposed a system based on code evolution analysis and a clone detection technique to indicate cryptocurrencies that might be vulnerable.

In this paper, we aim to provide a comprehensive review of code clone detection methods based on vulnerable code analysis. We compare previous studies following several different analysis methods and detection techniques, and then discuss open questions and future research

trends in this field. Our contributions are summarized as follows:

- We illustrate some key terminology, including related definitions, code clone types, and techniques with code samples.
- We comprehensively review and introduce previous security-related studies following three classifications and various comparison criteria.
- We discuss some open questions about deep learning-based code clone vulnerability detection approaches, which are more advanced and adaptive to technological development than current technologies, and code clone detection for 5G-IoT devices and real-time detection methods that can detect clone attacks more efficiently.

To enable a better understanding of what we discuss in this paper, it is organized to answer the following questions:

1) What is a code clone and why does it make software vulnerable to cyberattacks?
2) What types of vulnerabilities can be detected by code clone detection mechanisms?
3) What types of technologies can detect vulnerable code clones and how do those technologies work?
4) What is the future research direction for vulnerable code clone detection?

The motivation of this paper is to find corresponding answers to above research question.

This paper focuses on the security perspective of code clone detection. Most studies are selected by searching keywords *code clone vulnerability detection* from e-resources, such as Elsevier and Springer, which are authentic and accepted with high impact factors. We also select research works which are published on international conferences and journals organized by ACM an IEEE. We selected 50 works from IEEE, 17 works from ACM, 7 works from Springer, 5 works from Elsevier, and 4 works form arXiv preprint. This paper does not discuss much of studies which are not regarding security issues of code clone detection, such as the general programming and maintenance issues aroused from code clones.

The remainder of the paper is organized as follows: In Section II, we provide background information on why code cloning occurs, and introduce several representative studies and definitions of related terminology. In Section III, we thoroughly review existing security-related code clone analysis methods and detection techniques, and compare previous studies. In Section IV, we discuss deep learning-based approaches, 5G-IoT-based code clone detection, and real-time detection research directions. In Section V, we conclude this paper and outline avenues for future studies

II. BACKGROUND

In this section, we introduce the following:

- general background about why code cloning occurs; that is, why programmers prefer to use copy-paste methods during their programming work;
- issues arising from code cloning behavior; and
- definitions of several code clone-related terms, types of code clones, and code clone detection phases and detection techniques.

A. WHY CODE CLONING OCCURS

A good software engineering project should be developed with thorough and mature programming; however, sometimes, programmers prefer to reuse a code fragment [16] to finish their tasks, even if this is not encouraged. We analyzed several reasons for code cloning.

1) COST AND TIME CONSTRAINTS

The main reason for code cloning is that it can help programmers to finish software development more efficiently by reducing cost and time, particularly when meeting task deadlines [2], [16], [17].

2) LIMITATIONS OF PROGRAMMERS' SKILLS

Some junior, and even senior, programmers may receive specific programming tasks beyond their capabilities. For example, they may lack programming language proficiency or have difficulty in understanding those tasks, which facilitates code reuse [18].

3) USE OF TEMPLATES

Increasingly, code templates are providing more thorough and mature code, algorithms, and frameworks for programmers to help them to finish software development more efficiently. However, programs that use the same template could include identical or closely similar code fragments, which leads to code clones [2], [16].

4) FEAR TO BRING IN NEW IDEAS

Sometimes, new ideas or fresh code may result in a lengthy software development life cycle, or even introduce new errors to existing software [19], [20]. Hence, programmers fear bringing new ideas or fresh code into their existing project [2].

5) ACCIDENTAL CLONING

Sometimes, a programmer writes a piece of code that accidentally matches existing code, which leads to a type of accidental code cloning [16].

B. ISSUES

As a result of the reasons for code cloning mentioned above, whether intentional or unintentional, code clones have led to some issues in software development and maintenance.

- **Maintenance cost:** Cloning a piece of code in software can increase the post-implementation maintenance effort. For instance, if one cloned code fragment is modified, all other cloned code fragments have to be located to maintain the consistency [21].
- **Bugs propagation:** Cloning a piece of code that includes a bug can propagate the bug to different locations in the software system [2], [16], which also increases the maintenance effort to identify this bug from all cloned code fragments.

- **Vulnerability propagation:** If a piece of code that is vulnerable to specific attacks is copied, it will lead to vulnerability propagation across the entire software system. In this paper, we mainly discuss vulnerability detection approaches in software/code clones.

C. CLONE DETECTION

Researchers have addressed clone issues by providing code clone detection tools or approaches. Baker [22] developed a program called *dup*, which can locate duplicate or near-duplicate code sections in large software systems. Kamiya *et al.* [23] proposed a token-based clone detection tool called *CCFinder*, which extracts code clones in C, C++, Java, COBOL, and other source files. Li *et al.* [19] proposed a tool called *CP-Miner*, which can identify cloned code fragments in large software systems using data mining techniques. Roy and Cordy [24] proposed a lightweight application called *NiCad* for a source code transformation system that can find near-miss clones by applying an efficient text line comparison technique.

Several studies have also been conducted on clone detection approaches based on the application level. For Android applications, Crussell *et al.* [25] presented a detection tool called *DNADroid*, which can identify cloned applications by computing the similarity between two applications. Chen *et al.* [26] proposed an approach to measure the similarity between code sections in two applications on the basis of the method level. Similarly, Akram *et al.* [27] designed *DroidCC* for detecting cloned Android applications based on the source code level.

Meanwhile, many researchers have conducted excellent surveys on the topic of code clone detection. Rattan *et al.* [2] provided an extensive systematic literature review of software clones in general and software clone detection in particular, based on reviewing 213 articles from 2,039 articles published in 48 publication resources. Sheneamer and Kalita [1] discussed details of code clones, such as types of clones, detection phases of clones, detection techniques and tools, and challenges faced by clone detection techniques, by analyzing previous related studies. Saini *et al.* [16] discussed code clone detection and management to help researchers to start quickly on the basic concept of code clones and detection techniques. Ain *et al.* [3] provided a comprehensive review of the latest code clone detection tools and techniques, and a systematic literature review of 54 studies.

In this paper, we focus on providing a comprehensive literature review of vulnerability detection approaches in code clone areas to help to provide researchers with a clearer direction for future studies.

D. TERMINOLOGIES

We summarize several terms related to code cloning to help readers to obtain a basic understanding of code clones [1], [16], [28].

1) CODE FRAGMENT

A code fragment is a piece of source code, with or without comments, in a software project. It can contain any number of lines, statements, begin-end blocks, methods, or functions needed to run a program. For instance, *CODE FRAGMENT 1-5* are five different code fragments.

Code Fragment 1 Original Code Fragment

```
Data: A string
Result: Count the number of one certain letter in the string
def countElem (string, elem):
    num = string.count(elem, 0, len(string))
    # comment 1
    print(num)
stri = "Hello world!"
sub = "l"
countElem(stri, sub)
# comment 2
```

2) CODE CLONE PAIR

If a code fragment is identical or similar with minor modifications to another code fragment, which means that they are code clones, these two code fragments are called a code clone pair. For example, the pair that consists of *CODE FRAGMENT 1* and *CODE FRAGMENT 2* is a code clone pair.

3) CLONE CLASS

A clone class refers to a set of code clone pairs (more than two code fragments) related to each other, with the same equivalence relation. *CODE FRAGMENT 1-5* could be a clone class.

4) CLONE GRANULARITY

Clone granularity can be regarded as a research or detection level. This means that the detection method can be executed at the level of, for example, functions, classes, blocks, statements, and files. Granularity can be predefined for directional detection or not predefined, as for free granularity clones.

5) PRECISION AND RECALL

Precision and recall are two critical factors for evaluating the system accuracy of detecting software clones. Precision refers to the percentage of true negatives detected, and recall refers to the percentage of total clones detected in the software system, including false positives.

E. TYPES OF CLONES

To better understand what type of study belongs to code cloning and analyze the target source code more efficiently, the code clone issue could be classified into two main groups: textual-level clone and semantic-level clone [1], [29], [30]. *CODE FRAGMENT 1-5* provide five code fragments (*Python*) as examples of these two groups of clones. *CODE FRAGMENT 1* is an example of the original fragment.

1) TEXTUAL-LEVEL CLONE

This type of clone refers to two code fragments that perform almost the same text task [31]. For a textual-level clone, this can be further classified into three types of clone.

Code Fragment 2 Type-1 Exact Clone

```
Data: A string
Result: Count the number of one certain letter in the string
def countElem (string, elem):
    num = string.count(elem, 0, len(string)) # comment 1
    print(num)
stri = "Hello world!"
sub = "l"
countElem(stri, sub) # comment 2
```

Type-1: Exact clone A code fragment that is almost an exact copy of the original code fragment except for whitespace, blanks, and comments is regarded as an exact clone. Compared with the original code fragment, the Type-1 code fragment simply modifies the layout of comments and deletes one blank line, so it clearly belongs to the exact clone type.

Type-2: Renamed clone A code fragment that is similar to the original code fragment except for the names of variables, functions, types, and literals is regarded as a renamed clone. As shown in *CODE FRAGMENT 3*, compared with the original code fragment, the Type-2 code fragment modifies the function's name from 'countElem' to 'num_of_string,' and some variables, such as 'string' to 'a' and 'elem' to 'b.'

Code Fragment 3 Type-2 Renamed Clone

```
Data: A string
Result: Count the number of one certain letter in the string
def num_of_string (a, b):
    number = a.count(b, 0, len(a))
    # comment 1
    print(number)
string = "Hello world!"
letter = "l"
num_of_string(string, letter)
# comment 2
```

Type-3:Near miss clone A code fragment that is almost the same as the original code fragment except for some modifications, such as added or removed statements, and a different use of literals, variables, layout, and comments [1]is regarded as a near miss clone. The Type-3 code fragment belongs to the near miss clone type because of the modification that only replaces variables 'string' and 'elem' with 'a' and 'b,' respectively.

2) SEMANTIC-LEVEL CLONE

The second group of code clones is based on the semantic level, and is called the Type-4 clone.

Type-4: Semantic clone A code fragment that is similar to the original code fragment based on their functions and not syntax [32]is referred to as a semantic clone. The Type-4 code fragment modifies the code using 'for loop' to implement the same result achieved using the function 'count,' which refers to a semantic clone.

Code Fragment 4 Type-3 Near Miss Clone

```
Data: A string
Result: Count the number of one certain letter in the string
def countElem (string, elem):
    a = string
    b = elem
    num = string.count(b, 0, len(a))
    # comment 1
    print(num)
stri = "Hello world!"
sub = "l"
countElem(stri, sub)
# comment 2
```

Code Fragment 5 Type-4 Semantic Clone

```
Data: A string
Result: Count the number of one certain letter in the string
def countElem (string, elem):
    num = 0
    for i in string:
        # comment 1
        if i == 'l':
            num = num + 1
    print(num)
stri = "Hello world!"
sub = "l"
countElem(stri, sub)
# comment 2
```

F. CODE CLONE DETECTION PHASES

Fig. 1 shows the entire life cycle of code clone detection; some researchers prefer to call the process from *pre-processing* to *report clones* clones the code clone detection phases. The *Code clone detector* is the main component of a clone detection system, and is in charge of acquiring copy-pasted or duplicated source code and then processing the major clone detection phases. For instance, Davey *et al.* [33] provided a comprehensive illustration of the fundamental process of developing SOM-based and DCL-based clone detection tools.

1) PRE-PROCESSING

Pre-processing is the first step of code clone detection that [28]:

- removes all uninteresting or irrelevant parts of the source code, such as whitespace and comments, to reduce unrelated comparisons and calculation;
- identifies the remaining source code as source units, which are used for checking for the existence of direct clones' relations to each other after removing irrelevant fragments [1]]; and
- divides sources units into smaller comparison units depending on the comparison algorithm.

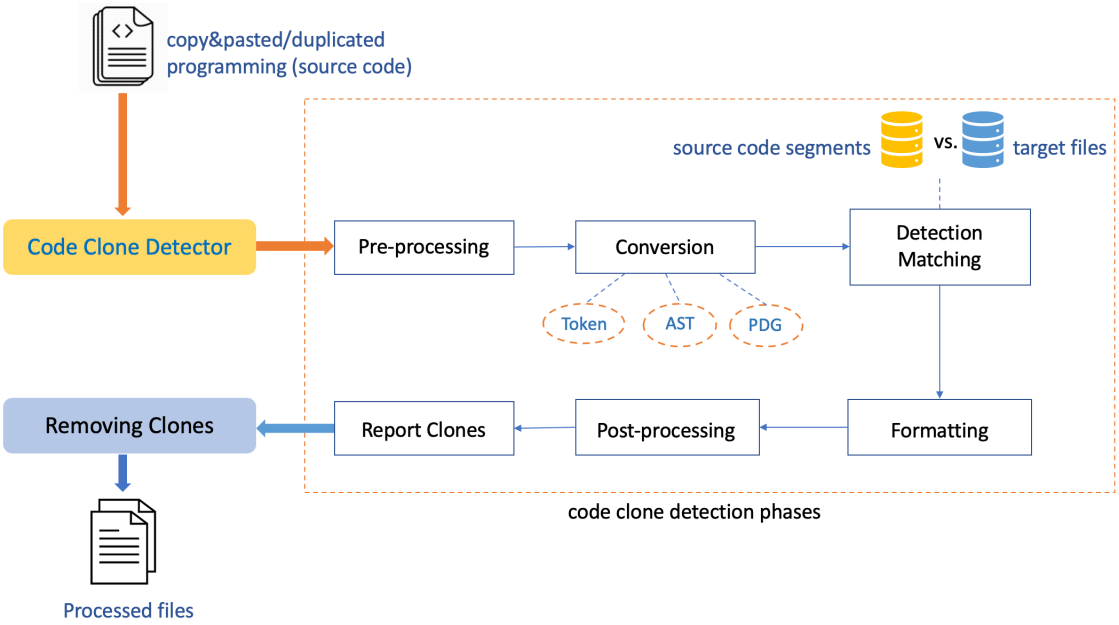


FIGURE 1. Life cycle of code clone detection.

2) CONVERSION

Conversion, also called *transformation*, is used to convert the source code acquired from the pre-processing step into a corresponding intermediate representation for further comparison [16]. Types of intermediate representation are the *Tokens*, *Abstract Syntax Tree* and *Program Dependency Graph*, which we introduce in detail in Section III.

3) DETECTION MATCHING

This step compares the source code units with target files using a particular comparison algorithm to identify similar source code fragments. The output of this step is a list of clone pairs or clone classes.

4) FORMATTING

This step formats the list of clone pairs obtained from the previous step based on the comparison algorithm into a new clone pair list related to the original source code.

5) POST-PROCESSING

Post-processing, also called *filtering/manual analysis* [34]], is not required by all code clone detection systems, and is used to filter out false positives or missed clones on the basis of reanalysis by human experts or automated heuristics.

6) REPORT CLONES

Clone results analyzed and confirmed by previous detection phases can be reported to the system for further action, such as correcting or removing the source code.

III. SECURITY-RELATED WORKS

In addition to the maintenance and debug cost arising from code clone behavior, software vulnerability propagation is another serious issue. Programmers may use source code files downloaded from websites that are intensively modified by attackers that can help those attackers to infiltrate their target systems easily. Islam *et al.* [35] proved that the security vulnerabilities found in code clones have a higher severity of security risk than those in non-cloned source code by detecting code clones and vulnerabilities in 8.7 million lines of code over 34 software systems based on quantitative analysis with statistical significance. Karademir *et al.* [36] conducted an experiment that used the *NiCad* [37] clone detector to identify JavaScript vulnerabilities in PDF files. Nappa *et al.* [38] presented a systematic study of the effect of shared/cloned code on vulnerability patching for client-side applications.

In this section, we provide a comprehensive review of recent security-related studies, analyze and discuss their primary purpose, present systems or architectures, and evaluate results from different analysis methods and detection techniques.

A. STATIC ANALYSIS VERSUS DYNAMIC ANALYSIS

The code clone detector shown in Fig. 1 plays an essential role during the entire life cycle of clone detection. Furthermore, analysis methods can be regarded as core functions of clone detectors. Analysis methods of code clone detection can be classified as *static analysis* and *dynamic analysis*, in addition to *hybrid analysis*, which refers to the advanced combination of both.

1) STATIC ANALYSIS

Static analysis refers to analyzing a piece of source code to detect possible defects in the early stage without any program's dynamic execution. Two types of static code analysis methods exist: one uses a machine that can read and check the source code automatically to detect possible clones, and the other is performed by a human reviewing the source code, also called code review [39]. The reviewer could be an expert or peer developer who fully understands the source code and manually reviews it to identify any missed clones or false positives.

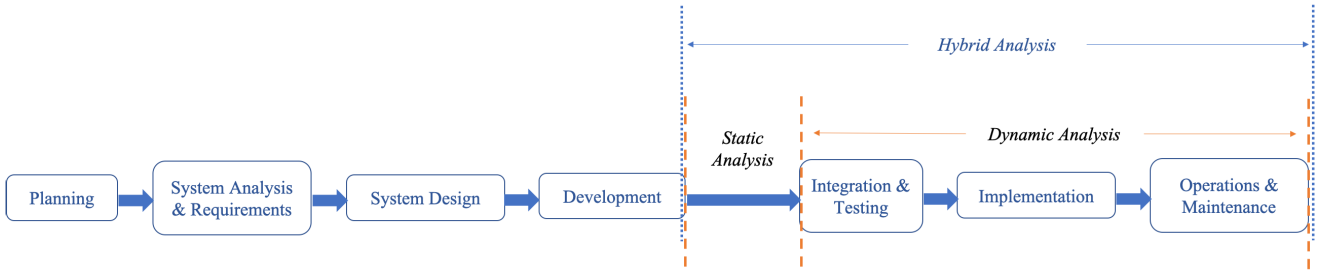


FIGURE 2. Possible analysis methods during a system development life cycle.

a: VULNERABILITY DETECTION VERSUS CODE CLONE DETECTION

Static code analysis is typically used for software/source code vulnerability detection. Source code vulnerability detection methods normally refer to the code or function similarity comparison between detected and target files based on normalizing or abstracting the source code into a representation. Code clone vulnerability detection can be regarded as a type of special software vulnerability detection, where the original code fragment is the target code fragment. Detection methods, particularly for vulnerability identification, can be adopted as references in the detecting vulnerable code cloning scenario. In this section, we discuss both of vulnerability detection and code clone detection.

Table 1 provides a comparison of several vulnerable code clone detection studies based on static code analysis methods. It compares these studies by illustrating their primary research purposes, detection techniques, and evaluation factors.

i) Source Code Vulnerability Detection: Zhang *et al.* [40] proposed an approach that uses trace-based security testing methods to detect software vulnerabilities in C programs. They generated a program constraint (PC) and obtained a security constraint (SC) by applying symbolic execution based on each hotspot mentioned above. The judgment condition for a vulnerable hotspot was $PC \wedge \overline{SC}$, which means it satisfies PC but violates SC.

To enhance the accuracy of vulnerable source code similarity analysis, Zhu *et al.* [44] proposed a solution that combines the Simhash algorithm and MD5 matching algorithm. The authors considered the problem in which the traditional hash algorithm cannot record the difference between similar files by generating identical fingerprints with local sensitive hashing. They used the Simhash algorithm to complement the file-level homology analysis algorithm based on MD5 matching.

As mentioned previously, automatic static analysis has some limitations, such as missing checks in the source code. Yamaguchi *et al.* [11] introduced a method called *Chucky*that can identify missing checks (for vulnerability discovery) in the source code automatically based on static analysis to help to accelerate the manual code review. Their method includes five major steps: (a) extract sources, sinks, conditions, assignments, and API symbols from a function's source code using a robust parser; (b) identify functions in which a similar context code operates; (c) determine only those checks associated with a given source or sink; (d) embed a selected function and its neighbors in a vector space using the tainted conditions; and (e) perform anomaly detection for missing checks based on identifying large distances from the normality model over the functions. They also provided suggestions for correcting potential fixes.

ii) Vulnerable Code Clone Detection: Jang *et al.* [41] proposed a detection system called *ReDeBug*that focuses on detecting unpatched source code flaws from code cloning. Unpatched code clones refer to buggy codes that are cloned by programmers but missed or unpatched when patches to source files are debugged and installed. Compared with previous detection techniques, *ReDeBug* does not focus on the number of detected clones but the scalability across the entire operating system. *ReDeBug* performs as a language-agnostic system to identify sequences of known vulnerable patched code fragments that are extracted and normalized from the *diff* files in the source code file to obtain the unpatched code clone list.

Li *et al.* [42] proposed a software vulnerability detection system by applying a backward trace analysis approach and symbolic execution method. Their system considers only vulnerability-related paths to mitigate the path exploration problem. They implemented this system using backward tracing of sensitive data used in a detected hotspot. They then used a data flow tree to recover the program's execution paths, which helped them to focus only on sensitive related data. Like Zhang *et al.*'s study, they also applied PC and SC mechanisms to verify existing vulnerabilities. They also proposed a software vulnerability discovery mechanism using code clone verification *(CLORIFI)* [6]], which can discover vulnerabilities in real-world programs in a scalable manner.

Kim *et al.* [43] proposed a scalable approach called *VUDDY* for code clone vulnerability detection. Its extreme scalability is achieved by leveraging function-level granularity and a length-filtering technique to reduce the number of signature comparisons. Their approach was divided into two main sections: pre-processing and clone detection [47].

- The pre-processing section includes retrieving functions from a given program using a robust parser, abstracting the source code by replacing it with symbols, normalizing the code body by removing unnecessary parts, and generating fingerprint dictionaries for the next detection process.

TABLE 1. Comparison of static analysis-based code/clone vulnerability detection.

Authors	Year	Code Clone Related	Purpose	Technique	Manual Code Analysis	Evaluation	Ref.
Zhang et al.	2010	No	Detecting vulnerabilities in C programs.	*SecTAC*: Trace-based symbolic execution by judging program constraint and security constraint and satisfiability analysis.	Need	Quickly detected all reported vulnerabilities and 13 new ones that have not been detected before from applied programs (14 benchmark and 3 open-source).	[40]
Jang et al.	2012	Yes	Detecting unpatched vulnerable code clones in the entire operating system distributions.	*ReDeBug*: Identifying sequences of known vulnerable patched code fragments, which is extracted and normalized from the *diff* files, in the source code file to get the unpatched code clone list.	No	1) Language agnostic. 2) Compared to previous work, better scalability with decreased false rate. 3) Found 15,546 unpatched vulnerable copies. 4) Confirmed 145 real bugs in the real world.	[41]
Li et al.	2013	Yes	Detecting software clones by selecting only vulnerability related paths.	Backward trace analysis & symbolic execution by judging program constraint (PC) and security constraint (SC).	Need	1) Precision: 83.33% 2) Recall: 90.90% 3) FI_Value: 86.95%	[42]
Yamaguchi et al.	2013	No	Detecting missing clones (vulnerability discovery) to help accelerate the manual code reviewing.	*Chucky*: Parsing source code to extract sources and sinks and then identifying similarities.	Need	Identified 12 missing cloning vulnerabilities in two (Pidgin and LibTIFF) of five projects (Firefox, Linux, LibPNG, LibTIFF and PidPNG).	[11]
Kim et al.	2017	Yes	Detecting vulnerabilities in a large software programs and reducing the number of signature comparisons.	*VUDDY*: Generating fingerprint dictionary through abstracting and normalizing; comparing fingerprint of vulnerabilities with target programs.	No	1) Able to process a billion lines of code in 14 hours and 17 minutes. 2) Detected 24% more unknown vulnerable clones.	[43]
Zhu et al.	2019	No	Enhancing the accuracy of code vulnerability similarity analysis.	Combining the Simhash algorithm and MD5 matching algorithm based on utilizing LSH to generate similar fingerprints.	No	Tested in three different projects with all precision values above 95%.	[44]
Bowman & Huang	2020	Yes	Identifying modified vulnerable code clones and all types of clones.	*VGRAPH*: Comprising the code property relationships between three graph-based components (triplet match) extracted from the contextual code,the vulnerable code, and the patched code.	No	1) Compared with four vulnerability detection techniques: FlawFinder, RATS, VUDDY, ReDeBug. 2) Precision: 98% Recall: 97% $F1_Value$: 97%	[45]
Mishra et al.	2020	Yes	Defending against code reuse attacks with specializing functions with a restricted interface.	*Saffire*: function specialization based on static argument binding and dynamic argument binding.	Need	1) Addressed dynamic arguments input limitations while system running. 2) Low runtime and memory overhead. 3) Could prevent whole-function reuse in critical system functions.	[46]

- The detection section works by comparing the fingerprint dictionary of vulnerabilities with the fingerprint dictionary of target programs by applying key lookup and hash lookup algorithms.

Bowman and Huang [45] used software source code properties to implement a more robust vulnerable code clone detection system called *VGRAPH*. Their system aims to identify vulnerable code modification and all types of

clone attacks by comprising the code property relationships between three graph-based (code property graph) components extracted from the contextual code, vulnerable code, and patched code. They called it a *Triplet Match*. To evaluate their detection technique, Bowman and Huang also compared *VGRAPH* with four state-of-the-art vulnerability detection techniques, that is, *FlawFinder*, *RATS*, *VUDDY* [43], and *ReDeBug* [41]], in accordance with the true positive, false positive, false negative, precision, recall and F1 values.

Another scenario may lead to missing clones, that is, the dynamic argumentation of source code functions. Normally, static code analysis focuses on the static arguments of source code functions, and then the dynamic arguments passed to the source code functions are ignored while the system is running. Mishra and Polychronakis [46] recently presented a compiler-level defense approach called *Saffire* against code clone/reuse attacks. *Saffire* performs static code analysis by eliminating the static arguments and restricting the acceptable dynamic values of arguments (user input, file address, and system status) during system runtime. This approach applies a narrow-scope form of data flow integrity to specialize functions with a restricted interface.

Although static code analysis is efficient in the early stage of the code clone detection life cycle, there are some inevitable limitations, such as time consumption, personnel training, and vulnerabilities introduced during program runtime. Goseva-Popstojanova and Perhinschi [48] evaluated three widely used commercial static code analysis tools to detect security vulnerabilities based on C/C++ and Java programs. Their experiment showed that a certain number of vulnerabilities were missed by all three tools. Furthermore, they did not provide any assurance of software product security and required further manual effort to classify reported warnings. Hence, dynamic analysis is needed for the late stages, particularly unit testing.

2) DYNAMIC ANALYSIS

Opposite to static analysis, dynamic analysis is performed by executing the program with real-time data to detect target system cloning issues [39]. Dynamic analysis can proceed on virtual machines, or even real processors, by monitoring the system's behavior while the system is running. This type of analysis method helps to detect vulnerabilities introduced during the entire system life cycle, particularly after static code analysis.

A critical role of dynamic analysis is to detect the real-time vulnerability introduced to avoid missing clones during the entire system life cycle. It is not easy to provide an explicit definition of dynamic analysis. Some researchers analyze application similarity on a code/method/function level, but we classify this kind of analysis as dynamic analysis after the implementation phase. In Table 2, we summarize some dynamic analysis studies for clone attack detection based on various working environments with corresponding attack methods, technologies, and evaluations.

a: SENSOR NETWORKS

Sensor networks provide a vulnerable environment for adversaries to easily compromise and duplicate sensors, and use them as weapons to obtain access to the entire network using legitimate credentials [50]. Parno *et al.* [51] presented a detection system to prevent the node replication attack in a distributed sensor network environment. However, their study did not mention further attacks that result from cloning compromised sensors that spread to the entire network. Choi *et al.* [49] provided a clone detection scheme called *SET*in sensor networks. They modeled a sensor network as a set of non-overlapping sub-regions, and assigned a unique identifier to each sensor node. The subset of each node in each sub-region is exclusive to other nodes. If adversaries capture, compromise, and duplicate sensor nodes in the network, the clone attack can be detected because of the intersecting subsets of the cloned nodes. Xing *et al.* [50] proposed an approach for the real-time detection of cloned-sensor attacks in wireless sensor networks by computing the fingerprint of each sensor to extract the neighborhood characteristics and check the validity of the originator's fingerprint for each message. Their approach achieved high detection accuracy based on a low computation and storage cost for node/sensor cloning scenarios during fingerprint generation and the detection phase. Furthermore, with no limitation on the number of cloned sensors, their approach improved on the results of related studies [49], [51].

b: INTERNET OF THINGS

The rapid development of the IoT has triggered many security issues, including various malicious code injections into IoT devices. Program developers prefer to use the software clone method to finish tasks quickly because of the large scale and range of IoT devices. The consequent clone attacks need more efficient corresponding detection approaches. To detect code clones in IoT applications, Tekchandani *et al.* [53], and Luo *et al.* [54] provided good results based on semantic-level source code analysis; however, their studies were not primarily on cloned vulnerability detection. Sachidananda *et al.* [55] proposed a framework to detect various vulnerabilities located in IoT devices using the static analysis method. Their approach was efficient in terms of identifying many types of attacks, such as memory leaks, code injection, buffer overflow, and other code-related vulnerabilities. Liu *et al.* [56] also proposed a similar vulnerability detection method for IoT binary code, but not for code clone attack detection in particular.

Gao *et al.* [14] presented an approach called *IoTSeeker* for cross-platform IoT device vulnerability detection based on analyzing binary code at the semantic level. They constructed a labeled semantic flow graph to capture both data flow and control flow information from binary code. They then extracted semantic features as numerical vectors and built a detection neural network model for feature integration and vulnerability search. Finally, *IoTSeeker* calculated the cosine distance between two embedding vectors to identify whether vulnerable clones exist.

TABLE 2. Comparison of dynamic analysis-based software clone attacks detection.

Authors	Year	Environment	Attack Method	Real-time detection	Technique	Evaluation	Ref.
Choi et al.	2007	Sensor networks	Cloned & compromised sensors	No	*SET* operations on *subset trees* of all sensor nodes to identify whether intersecting subsets (code clones) exists or not.	1) Low communication cost (O(N)) 2) Memory overhead need further reduction	[49]
Xing et al.	2008			Yes	Checking each sensor's fingerprint.	1) High detection accuracy 2) Low communication/ computation/storage cost 3) Improvement on number limitation compared to [51], [49]	[50]
Crussell et al.	2012	Android applications	Type-1, type-2, type-3 clones and injected vulnerabilities during runtime.	No	*DNADroid*: Detecting 1 clone attacks by comparing program dependency graphs based on method-level.	1) Low false positive rate with identifying indeed similar clones. 2) May overlook cloned applications. 3) Need further comparison to other approaches.	[25]
Chen et al.	2014			No	*Centroid*: To measure the similarity between code fragments (method-level) in two applications based on 3D control flow graphs.	1) Less than one hour for cross-market app clone detection. 2) High accuracy for distinguishing cloned methods. 3) Further improvement for detecting Type-4 clones. 4) Not efficient for partial cloning.	[26]
Akram et al.	2018			Yes	*DroidCC*: Detecting APK source files based on excluding third-party libraries, normalization and feature extraction.	1) Low cost, reliable and scalable. 2) High accuracy of 87%.	[27]
Gao et al.	2019	IoT devices	Compromised/ cloned IoT devices	Yes	*IoTSeeker*: labeled semantic flow graph, semantic feature extraction, neural network model, calculating vectors distance.	High accuracy for both identifying code clone and vulnerability search.	[14]
He et al.	2019	Ethereum Smart Contract	Cloned contact code	Yes	Comparing similarities between generated fingerprints of user-created contracts code and contract-created contracts code in EVM run time to identify vulnerable code clones.	Successfully identified over 53% similar contract pairs have the same vulnerability behaviors and over 46% have different vulnerability behaviors.	[52]
Hum et al.	2020	Cryptocurrency	Propagating vulnerable cloned cryptocurrency	No	Comparing given reported vulnerability with target cryptocurrency based on code evolution analysis and a clone detection technique.	By answering: 1) clone prevalence 2) approach accuracy 3) true positives vs. false positives.	[15]

The supply chain provides another platform for introducing software clone attacks, such as cloned and compromised RFID tags, which may help attackers to acquire confidential credentials and authorization information to compromise the supply chain system. Researchers [57]–[60] have proposed several clone detection approaches for an RFID-embedded

supply chain system, and these approaches can be applied to vulnerable RFID tag clone detection with the appropriate improvement.

c: ANDROID APPLICATIONS

The Android operating system has become more popular and widely used, and more security concerns have attracted researchers' attention. Some researchers have detected source code similarities for Android applications, including Type-1, Type-2, and Type-3 clones, and also injected vulnerabilities into applications during software runtime. Crussell *et al.* [25] presented a cloning attack detection tool called *DNADroid*, Chen *et al.* [26] presented a similarity/clone detection approach called the *centroid*, both which are based on comparing program dependency graphs between methods in candidate applications. Crussell *et al.*'s study focused only on identifying similar clones, thereby leading to a low false positive rate and missing clones. Chen *et al.*'s approach is more accurate, has the explicit purpose of improving the detection system's accuracy and scalability, and has a greater focus on cross-platform application clone detection. Akram *et al.* [27] proposed a scalable clone detection approach called *DroidCC* based on excluding third-party libraries, normalization, and feature extraction, and evaluated their approach on a real-time dataset.

d: ETHEREUM SMART CONTRACT

With the rapid development of blockchain's distribution architecture, the Ethereum smart contract provides an environment for malicious code clones by injecting a piece of contract code and propagating it to other blocks. He *et al.* [52] focused on the ecosystem of the Ethereum smart contract to characterize vulnerable code clones using the *fuzzy hashing* technique to calculate the edit distance between two fingerprints. Their approach compares the similarity between generated fingerprints of user-created contract code and contract-created contract code during Ethereum virtual machine runtime.

e: CRYPTOCURRENCY

Cryptocurrency is another research topic of great interest because of its novel security protection structure and wide use in both academic research and industrial applications. Hum *et al.* [15] proposed an approach called *CoinWatch* for detecting code/system vulnerabilities in cryptocurrencies on the basis of code clone detection technology. They provided this type of approach because of the rapidly increasing use of cryptocurrencies (e.g., Bitcoin) and their publicly readable code structures [61], [62]. If one code fragment is vulnerable to cyberattacks, the vulnerability is propagated into other cloned code fragments or even cryptocurrencies.

CoinWatch has four main phases for vulnerabilities detection:

- **CVE parsing & linking it with commits:** The first phase involves CVE parsing and linking the result with possible commits. A target CVE is provided at input together with data publicly obtainable from its structured details [63]. After selecting a target CVE, CoinWatch performs code evolution analysis of the parent project to obtain bug fixing and bug introducing commits.
- **Identification of vulnerable code:** The bug introducing and fixing commits are then manually annotated to minimize the code responsible for the vulnerability and improve the program.
- **Initial filtering:** This phase can be regarded as pre-processing before moving to the detection process. To narrow down the search space, which means to work more efficiently, CoinWatch filtered the list of monitored projects on the basis of the fork's date before running the clone detector.
- **Detection process:** The last phase is the core part of CoinWatch: the clone detector. This part reports the cloned projects that are likely to be affected by the vulnerability given the filtered source code of the monitored cryptocurrencies.

Authors evaluated their approach by answering three research questions about clone prevalence in cryptocurrencies, the accuracy of CoinWatch, and the comparison of true positives with false positives in the vulnerability detection report.

3) DISCUSSION

Malicious people typically target web applications as an easy and flexible environment for code and script injection. However, few researchers have discussed this related clone problem. Vineetha and Krishna [65] researched this topic for code clone vulnerability analysis and detection in web applications by analyzing the web page structure and comparing the similarities. They did not propose a powerful detection system, and further evaluation for their approach is needed. Agrawal *et al.* [66] presented a detection framework to identify web application clones based on the source code level. They presented their framework following a detailed process introduction involving executing and monitoring, classifying and controlling, and refining and managing code. Many security practitioners have adapted this framework; however, the framework is limited to the source code level, which is not flexible for dynamic detection.

Following recent technological improvements, static or dynamic analysis per se cannot satisfy the requirement to prevent various increasing cyberattacks. For example, it is difficult and will take a longer time to trace back a piece of vulnerable code to its exact location through dynamic analysis only. Static analysis cannot obtain access to some types of source code files if the source code is not available or the executable file has been packed by packer or protector tools. Hybrid and advanced analysis methods, such as binary code-level detection methods, are necessary for more efficient code clone detection and source code fixing.

B. REPRESENTATIONS

Software clone detection techniques can be classified on the basis of different representations as five types: text-based,

TABLE 3. Comparison of detection-techniques-based code clone vulnerability detection.

Authors	Year	Representation	Clone Type	Purpose	Technique/Tool	Evaluation	Ref.
Jiang et al.	2007	AST	Type-1 Type-2 Type-3	Identifying code cloning with minor modification.	*DECKARD:* to cluster characteristic vectors by comparing subtrees similarities.	1) Detected more clone lines than CP-Miner. 2) Faster than CP-Miner. 3) Language independent.	[10]
Crussell et al.	2012	PDG	Type-4	Detecting vulnerable clones in Android applications.	Same as TABLE 2, entry 3.		[25]
Karademir et al.	2013	XML file	Type-3	Detecting near-miss clones in a set of PDF files.	NiCad clone detector	A small training set produced 87% detection of previously known malware with 1% false positives.	[36]
Chen et al.	2014	PDG	Type-4	Detecting vulnerable clones in Android applications.	Same as TABLE 2, entry 4.		[26]
Frahadi et al.	2015	Token	Type-1 Type-2 Type-3	Identifying code clones of a target malware from a collection of previously analyzed malware binaries.	*ScalClone*: applying two assemble code clone search methods for malware analysis at different token normalization levels.	1) Support large-scale assemble code search. 2) Effectively identifying assemble code clones for real-life scenarios.	[67]
Unruh et al.	2017	AST	Type-1 Type-2 Type-3	Identifying vulnerabilities from tutorial-style snippets.	A semi-automated approach by applying AST-based graph traversals to verify similarities in code snippets.	1) Identified 117 vulnerabilities that have a strong syntactic similarity to vulnerable code snippets. 2) Able to support large-scale clone discovery.	[64]
Alalfi et al.	2018	XMI textual representation	Type-3	Identifying near-miss clones in reverse-engineered UML sequence diagrams to characterize and abstract the run-time behaviour of web applications.	NiCad clone detector	Recall rate with 100% and precision rate with 86% compared to the previously published and validated SecureUML model [69].	[68]
Akram et al.	2019	Token	Type-1 Type-2 Type-3	Detecting vulnerable code clone in unpatched source code.	*VCIPR*: token-based approach for feature extraction at function level granularity.	Detecting unknown vulnerable clones with high accuracy of almost 83%.	[70]
Shi et al.	2019	AST	Type-1 Type-2 Type-3	Finding vulnerable OS code clones.	A two-phase approach for comparing correlations by extracting function features derived from the AST structure at the function level.	Improvement and higher accuracy compared to VUDDY and LSTM.	[71]

token-based, AST-based, program data graph-based, and metric-based. In this section, we provide a review of security analysis studies on these clone detection techniques. In Table 3, we summarize several studies by comparing their representations, purpose, possible detected clone types, applied techniques, and evaluations.

1) TEXT-BASED

Text-based code clone detection technology simplifies the source code to a sequence of characters by removing unnecessary parts, such as comments, whitespace, and new lines, from the source code [72]–[74]. It compares the similarity between these character sequences individually, and then returns the matching results [28]. Text-based code clone detection can be used to detect Type-1 (exact clones), Type-2 (renamed clones), and Type-3 (near-miss clones) code clones, which are based on the textual level.

Karademir *et al.* [36] and Alalfi *et al.* [68] both presented approaches for detecting vulnerable near-miss clones (Type-3) based on a text-based technique. Karademir *et al.*'s approach identifies malware from JavaScript in Adobe Acrobat (PDF) files. It compares the similarity between collected PDF files that contain JavaScript malware and clear JavaScript. Their approach uses the *NiCad* clone

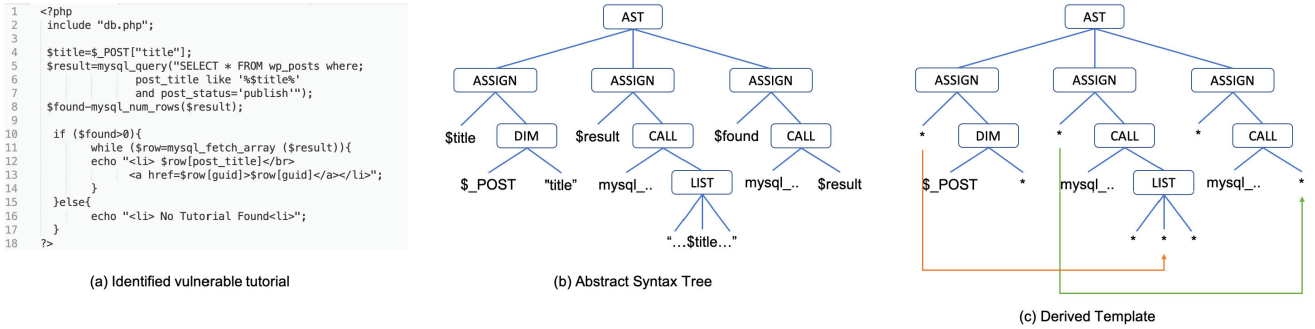


FIGURE 3. SQL Injection and XSS Code Samples and corresponding Abstract Syntax Tree and Derived Template [64]. This code snippet (a) contains an SQL Injection vulnerability occurs on line 6 as the variable $title will be posted in an SQL query without first security processing. Line 12 and 13 can result in the XSS attack by inserting database rows into the document directly. Part (b) describes an example of an abstract syntax tree generated from the SQL Injection vulnerability of part (a), in which leaf nodes correspond to identifiers (variables), API symbols or literals. Part (c) represents a derived template from the abstract syntax tree by replacing all variables and literals using wildcard symbols and introducing edges between nodes to represent the same variable.

detector, which is particularly for near-miss clone detection. Alalfi *et al.*'s approach identifies near-miss interaction clones in reverse-engineered UML sequence diagrams. Their approach works at the XMI level. They also used the *NiCad* clone detector to help to process the detection in the reverse-engineered behavioral model.

2) TOKEN-BASED

Token-based code clone detection technology converts the source code into an intermediate representation, that is, a token sequence, using a certain token conversion tool before the detection phase [1], [28]. One converted token sequence can be compared with another converted token sequence under a matching rule to obtain the matching results for further processing. Representative token-based techniques are *CCFinder* [23] and *CP-Miner* [75]. Compared with text-based techniques, a token-based technique is more robust against code changes, such as formatting and spacing [10].

Farhadi *et al.* [67] proposed a scalable code clone detection approach called *ScalClone* for malware analysis on the basis of their previous approach, which was for an assemble code clone detection method [12], [76]. Their approach discovers both exact and inexact clones at different token normalization levels using a large-scale assemble code search. Akram *et al.* [70] proposed a lightweight and scalable system called *VCIPR* for vulnerability detection in unpatched source code based on token normalization representation at function-level granularity. They built a fingerprint index of the top critical CVE's source code to detect unpatched code fragments in common open-source software.

3) TREE-BASED

Tree-based code clone detection technology also refers to AST-based technology [1]. In the code parsing process, the syntax tree-based method converts the source code into an AST, and the representation is the tree node before the matching and detecting phases [28]. The matching result is returned by comparing two converted syntax trees.

In the code clone area, a source program can be parsed into a parse tree or AST that represents the source code [10]. Subtrees can be compared through exact or close subtree matches to detect whether any code clones exist [77]–[79].

Unruh *et al.* [64] proposed an approach to semi-automatically detect vulnerable code snippets starting from certain web tutorials and QA websites, which aim at assisting programmers' coding tasks. They applied AST-based graph traversals to verify similarities in analyzed code snippets that correspond to the original vulnerability. Unruh *et al.* provided an example of an identified vulnerable code snippet taken from a popular PHP tutorial and its corresponding AST structure, and derived the template shown in Fig. 3. Shi *et al.* [71] proposed a two-phase framework (training phase and detection phase) to identify vulnerable source code clones in operating systems. The approach learns correlations on the basis of AST normalization at function-level granularity.

4) PDG-BASED

PDG-based detection technology refers to converting source code into a control flow and data flow graph, and then returning the matching result by comparing the similarities between the sub-graphs [28].

For Type-4 (semantic clones) code clones, the PDG-based code clone detection method is efficient in terms of detecting source code vulnerabilities because it preserves the semantic features of the program [28]. Several research studies have been conducted on the basis of this type of graph cooperation method for vulnerability detection.

The final subsection (dynamic analysis) introduces some studies that used program data graph abstraction for feature extraction. For instance, Crussell *et al.* [25] proposed *DNADroid*, Chen *et al.* [26] proposed the *centroid*, both for detecting Android application cloning vulnerabilities. Their approaches are capable of identifying Type-4 (semantic clones) code clones in a dynamic software operating environment.

5) METRIC-BASED

The metric-based code clone detection method parses the program by dividing the source code into several small code segments, and then calculates the difference value among these code segments and determines whether the calculated values are the same (a clone) [28]. Mayrand *et al.* [80] discussed using metric extraction techniques to automatically detect function cloning in a software system. Their study focused on analyzing and comparing control graph metrics and data flow graph metrics on the basis of a previous AST representation. Few researchers have primarily studied, or specially mentioned, applying metric-based detection techniques for vulnerable code clone detection. Therefore, a more in-depth survey is needed regarding this aspect.

6) DISCUSSION

As illustrated above, text, token, and AST-based detection techniques can identify textual-based clone attacks, and the PDG-based detection technique can detect semantic-based clone attacks. However, (i) few researchers have aimed to present a hybrid detection approach that is efficient in terms of detecting both textual and semantic-based code/software clone attacks; and (ii) it is not easy and obvious to select a normalized source code representation while designing a detection approach because there are no selection criteria.

C. BINARY-LEVEL DETECTION

When reviewing previous studies, we found that many researchers focused on analyzing binary code-based similarity comparison. The reason for binary-based analysis is that software source code cannot be acquired at any time. Because of some privacy protection reasons, researchers have to find another way to obtain software or application code, or function information. Another reason might be the huge task load of pre-processing, filtering, and feature extraction for source code information. Khoo *et al.* [81] provided a search system that identifies binary code by comprising instruction mnemonics, control flow sub-graphs [89]], and data constants extracted from binary code fragments. Lee *et al.* [83] introduced a method for identifying software vulnerabilities from assembly code using a deep learning mechanism. Hu *et al.* [82] presented a semantics-based approach to identify binary code clones.

Table 4 summarizes several binary-level-based code clone detection techniques following a set of specific criteria. As introduced in Table 4, some researchers have proposed efficient binary-code reuse analysis and detection methods. For instance, Frahadi *et al.* [12] introduced a method to identify malicious cloned code binaries based on the token normalization technique; Xue *et al.* [85] proposed a framework to detect vulnerable code clones by slicing binary codes and identifying domain-specific code fragments; Ishiura [87] proposed detecting the loss of guards by comparing binary-code pairs with or without problematic optimization; Ding [88] proposed learning lexical semantic relationships and the vector representation directly from plain assemble code instead of manually specifying it from prior knowledge; Liu *et al.* [56] proposed a long short-term memory (LSTM)-based approach to detect binary-level software vulnerabilities automatically.

However, binary-level analysis still faces several challenges. For instance, the limitation of accurately determining all valid control flow paths from the source code at system runtime and performing accurate static data flow analysis to identify argument values [46]. Mishra and Polychronakis [90] proposed Shredder for statically analyzing Windows applications at the binary level using backward dataflow analysis to derive expected argument values and generate application-wide policies for critical system functions. To address limitations in binary-level analysis, after *Shredder*, they proposed *Saffire* (section III). Hence, binary code-based clone attack detection is an important future research direction.

IV. FUTURE RESEARCH DIRECTION

For security analysis, several important topics on software clone detection remain, which we discuss here. Following the discussion in Section III, we summarize a potential research direction, which is an integration of *intelligent detection techniques, code clone detection for IoT devices and dynamic detection mechanisms*.

Some researchers have provided efficient results, for example, Gao *et al.* [14], and Liu *et al.* [56] proposed an in-depth learning-based approach for binary vulnerability detection at the semantic level for IoT devices. They trained a neural network model with numerical vectors transformed by the semantic features of the captured data flow and control flow information. We discuss three aspects of this type of research topic; however, we believe that there is a wider research space for this topic.

A. INTELLIGENT DETECTION

From Table 1, we found that some detection approaches based on the static analysis method partially relied on manually analyzing source code or generating representations, which typically takes time and effort, and is not efficient for solving the big-code problem [91]. Many researchers are moving toward applying more intelligent technology, such as deep learning and neural network models, to the research area of vulnerable source code detection [92], [93].

Kim *et al.* [94] used obfuscation techniques for obfuscated macro code detection based on training five machine learning classifiers and extracting 15 static discriminant features. Wang *et al.* [95] researched the patch level for "0-day" vulnerability detection by automatically identifying secret security patches in open-source software. They trained the identification model with extracted features from more than 4,700 security patches from a database to detect similar patches or vulnerabilities.

Li *et al.* [13], [96], [97] proposed deep learning-based approaches (*VulPecker, VulDeePecker, SySeVR*) for software vulnerability detection. Their approaches were aimed at

TABLE 4. Summary of binary-level code clone detection.

Authors	Year	Static Analysis	Dynamic Analysis	Vector Generation	Technique	Security	Evaluation	Ref.
Khoo et al.	2013	✓		Control flow sub-graphs.	*Rendezvous:* to identify code clones using control flow sub-graphs and data constants.	Not for vulnerability detection.	1. Time overestimate. 2. Need dynamic code instrumentation and symbolic execution. 3. Threats to validity. 4. F2-measure: 86.7%.	[81]
Hu et al.	2017	✓		Semantic signatures.	To measure similarities of binary functions by emulating executions and extracting semantic signatures.	Not for vulnerability detection.	1. Accuracy: 82.6%. 2. Higher efficiency of pre-processing and detection processing.	[82]
Lee et al.	2017		✓	*Instruction2vec.*	To compare vectored assembly code with training dataset to classify whether software weakness exists or not.	Improved.	Accuracy: 96.1%.	[83]
Farhadi et al.	2017	✓		Token normalization.	*BinClone*: to identify the code clone fragments from a collection of malware binaries based on the token-level.	Improved.	1. Precision rate: higher than 75%, recall rate: higher than 80%. 2. Avoided the non-deterministic issue as in LSH [84].	[12]
Xue et al.	2018	✓		Domain-related instructions.	*Clone-Slicer:* To detect code clones by slicing into and identifying domain-specific binary code fragments.	Not for vulnerability detection.	1. 43.64% improvement than Clone-Hunter [86]. 2. 32.96% time cut compared to Clone-Hunter.	[85]
Azuma & Ishiura	2019	✓		N/A	To detect the loss of guard by comparing a pair of binary codes generated from given source codes, with or without problematic optimization.	Can not detect vulnerability directly.	Detected 2 instances from 7 programs with one false positive.	[87]
Ding et al.	2019	✓	✓	Representation learning.	*Asm2Vec*: To jointly learn the lexical semantic relationships of assembly functions based on assembly code instead of manually specifying from prior knowledge.	Highly improved.	1. More resilient to code obfuscation and compiler optimizations. 2. 0 FP and 100% recalls for vulnerability detection.	[88]
Liu et al.	2020		✓	Binary function.	To detect vulnerable binary code using attention and LSTM mechanism.	Highly improved.	1. Accuracy: 83.93%. 2. Precision: 79.7%. 3. TPR: 90.41%. 4. FPR: 22.37%. 5. F1-measure: 84.72%.	[56]

automatically detecting vulnerable source code fragments by training a BLSTM neural network. They compared similarities between source code fragments and target vulnerabilities by generating *code gadgets* and transforming these *code gadgets* into vector representations, which were used as the neural network input. Their approach performed well in terms of finding vulnerabilities compared with similar systems, and was able to find many types of vulnerabilities simultaneously.

Although the series of *VulDeePecker* and Kim's study only focused on source code vulnerability detection based on similarity comparison, it was an efficient and applicable method for vulnerable code clone detection. The clone

detection system can be made more intelligent and automated by training it using the original vulnerable code fragments on the basis of deep neural network models and appropriate feature selection.

B. 5G-IoT DETECTION

The IoT network provides an environment for attackers to inject malicious code easily. IoT devices, particularly small devices, such as baby monitors, can be attacked easily by malicious code cloning without complicated or extensive code. The cloned vulnerability can spread in a moment through a network of a vast number of devices.

The global data volume is increasing, which makes 5G technology indispensable. For existing technologies, it is more challenging to meet the requirements of the rapidly developing IoT world. Next-generation technology, that is, 5G, will provide IoT devices with unlimited connectivity in the future internet world. Hence, code cloning is a primary challenge for 5G-IoT technology. Ullah *et al.* [98] proposed an approach to identify code clones in specific 5G-IoT applications using a control flow graph and deep learning model.

C. DYNAMIC DETECTION

From Table 2, we conclude that real-time clone attack detection is another possible research topic that needs further attention. Particularly for mobile applications and web environments, attackers can intrude on a running application at any time by executing malicious software clone behavior and controlling compromised applications. Applying real-time detection techniques to platforms at the application level is very necessary. As previously discussed, researchers have proposed code clone vulnerability detection approaches for IoT devices [14], [56], [99], and RFID-enabled supply chain systems [57]–[60]. Thus, a real-time cloning detection approach is needed to protect systems more efficiently.

V. CONCLUSION

In this paper, we provided a comprehensive review of previous studies on software/code clone detection from the security perspective. We compared and summarized several detection approaches based on static code analysis and dynamic analysis, respectively. Additionally, we outlined different representation-based studies and provided some meaningful information to researchers, such as possible detected clone types, the research purpose, and applied techniques or tools. We also discussed vulnerable code clone detection issues at the binary code level. Then we proposed a future research direction, including three potential topics, *intelligent detection, 5G-IoT-based clone detection and real-time detection*, which were generated from the literature review.

This survey provides a summary of previous vulnerable code clone detection-related results to help researchers to acquire basic knowledge of this topic, and select the correct techniques or tools while identifying potential research issues and future directions.

ACKNOWLEDGMENT

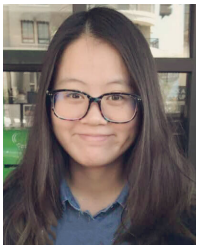
The authors thank Dr. Yujie Gu and all the reviewers for their helpful advice on this article. They also thank Dr. Maxine Garcia from Edanz Group (https://en-author-services.edanz.com/ac) for editing a draft of this article.

REFERENCES

[1] A. Sheneamer and J. Kalita, "A survey of software clone detection techniques," *Int. J. Comput. Appl.*, vol. 137, no. 10, pp. 1–21, Mar. 2016.
[2] D. Rattan, R. Bhatia, and M. Singh, "Software clone detection: A systematic review," *Inf. Softw. Technol.*, vol. 55, no. 7, pp. 1165–1199, Jul. 2013.
[3] Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool, "A systematic review on code clone detection," *IEEE Access*, vol. 7, pp. 86121–86144, 2019.
[4] S. Ducasse, M. Rieger, and S. Demeyer, "A language independent approach for detecting duplicated code," in *Proc. IEEE Int. Conf. Softw. Maintenance (ICSM), Software Maintenance Bus. Change*, Aug. 1999, pp. 109–118.
[5] R. Komondoor and S. Horwitz, "Using slicing to identify duplication in source code," in *Proc. Int. Static Anal. Symp.* Berlin, Germany: Springer, 2001, pp. 40–56.
[6] H. Li, H. Kwon, J. Kwon, and H. Lee, "CLORIFI: Software vulnerability discovery using code clone verification," *Concurrency Comput., Pract. Exper.*, vol. 28, no. 6, pp. 1900–1917, Apr. 2016.
[7] M. R. Islam and M. F. Zibran, "A comparative study on vulnerabilities in categories of clones and non-cloned code," in *Proc. IEEE 23rd Int. Conf. Softw. Anal., Evol., Reeng. (SANER)*, vol. 3, Mar. 2016, pp. 8–14.
[8] E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner, "Do code clones matter?" in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, May 2009, pp. 485–495.
[9] M. F. Zibran and C. K. Roy, "Conflict-aware optimal scheduling of code clone refactoring: A constraint programming approach," in *Proc. IEEE 19th Int. Conf. Program Comprehension*, Jun. 2011, pp. 266–269.
[10] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proc. 29th Int. Conf. Softw. Eng. (ICSE)*, May 2007, pp. 96–105.
[11] F. Yamaguchi, C. Wressnegger, H. Gascon, and K. Rieck, "Chucky: Exposing missing checks in source code for vulnerability discovery," in *Proc. CCS*, 2013, pp. 499–510.
[12] M. R. Farhadi, B. C. M. Fung, P. Charland, and M. Debbabi, "BinClone: Detecting code clones in malware," in *Proc. 8th Int. Conf. Softw. Secur. Rel.*, Jun. 2014, pp. 78–87.
[13] Z. Li, D. Zou, S. Xu, X. Ou, H. Jin, S. Wang, Z. Deng, and Y. Zhong, "VulDeePecker: A deep learning-based system for vulnerability detection," 2018, *arXiv:1801.01681*. [Online]. Available: http://arxiv.org/abs/1801.01681
[14] J. Gao, Y. Jiang, Z. Liu, X. Yang, C. Wang, X. Jiao, Z. Yang, and J. Sun, "Semantic learning and emulation based cross-platform binary vulnerability seeker," *IEEE Trans. Softw. Eng.*, early access, Dec. 2, 2020, doi: 10.1109/TSE.2019.2956932.
[15] Q. Hum, W. J. Tan, S. Y. Tey, L. Lenus, I. Homoliak, Y. Lin, and J. Sun, "CoinWatch: A clone-based approach for detecting vulnerabilities in cryptocurrencies," 2020, *arXiv:2006.10280*. [Online]. Available: http://arxiv.org/abs/2006.10280
[16] N. Saini, S. Singh, and S. Suman, "Code clones: Detection and management," *Procedia Comput. Sci.*, vol. 132, pp. 718–727, Jan. 2018.
[17] S. Wagner, *Software Product Quality Control*. Berlin, Germany: Springer, 2013.
[18] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in OOPL," in *Proc. Int. Symp. Empirical Softw. Eng. (ISESE)*, 2004, pp. 83–92.
[19] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-miner: Finding copy-paste and related bugs in large-scale software code," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 176–192, Mar. 2006.
[20] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *Proc. 6th Joint Meeting Eur. Softw. Eng. Conf. (ACM SIGSOFT), Symp. Found. Softw. Eng. (ESEC-FSE)*, 2007, pp. 55–64.
[21] C. J. Kapser and M. W. Godfrey, "Supporting the analysis of clones in software systems," *J. Softw. Maintenance Evol., Res. Pract.*, vol. 18, no. 2, pp. 61–82, 2006.
[22] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Proc. 2nd Work. Conf. Reverse Eng.*, 1995, pp. 86–95.

[23] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Softw. Eng.*, vol. 28, no. 7, pp. 654–670, Jul. 2002.
[24] C. K. Roy and J. R. Cordy, "NICAD: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization," in *Proc. 16th IEEE Int. Conf. Program Comprehension*, Jun. 2008, pp. 172–181.
[25] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on Android markets," in *Proc. Eur. Symp. Res. Comput. Secur.* Berlin, Germany: Springer, 2012, pp. 37–54.
[26] K. Chen, P. Liu, and Y. Zhang, "Achieving accuracy and scalability simultaneously in detecting application clones on Android markets," in *Proc. 36th Int. Conf. Softw. Eng.*, May 2014, pp. 175–186.
[27] J. Akram, Z. Shi, M. Mumtaz, and P. Luo, "DroidCC: A scalable clone detection approach for Android applications to detect similarity at source code level," in *Proc. IEEE 42nd Annu. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 1, Jul. 2018, pp. 100–105.
[28] H. Min and Z. L. Ping, "Survey on software clone detection research," in *Proc. 3rd Int. Conf. Manage. Eng., Softw. Eng. Service Sci. (ICMSS)*. New York, NY, USA: Association for Computing Machinery, 2019, pp. 9–16, doi: 10.1145/3312662.3312707.
[29] A. V. Aho, R. Sethi, and J. D. Ullman, "Compilers, principles, techniques," *Addison Wesley*, vol. 7, no. 8, p. 9, 1986.
[30] C. K. Roy and J. R. Cordy, "A survey on software clone detection research," *Queens School Comput.*, vol. 541, no. 115, pp. 64–68, 2007.
[31] C. K. Roy and J. R. Cordy, "A mutation/injection-based automatic framework for evaluating code clone detection tools," in *Proc. Int. Conf. Softw. Test., Verification, Validation Workshops*, 2009, pp. 157–166.
[32] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proc. 30th Int. Conf. Softw. Eng. (ICSE)*, 2008, pp. 321–330.
[33] N. Davey, P. Barson, S. Field, R. Frank, and D. Tansley, "The development of a software clone detector," *Int. J. Appl. Softw. Technol.*, vol. 1, nos. 3–4, pp. 219–236, 1995.
[34] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Sci. Comput. Program.*, vol. 74, no. 7, pp. 470–495, May 2009.
[35] M. R. Islam, M. F. Zibran, and A. Nagpal, "Security vulnerabilities in categories of clones and non-cloned code: An empirical study," in *Proc. ACM/IEEE Int. Symp. Empirical Softw. Eng. Meas. (ESEM)*, Nov. 2017, pp. 20–29.
[36] S. Karademir, T. Dean, and S. Leblanc, "Using clone detection to find malware in acrobat files," in *Proc. Conf. Center Adv. Stud. Collaborative Res. (CASCON)*. New York, NY, USA: IBM Corp., 2013, pp. 70–80.
[37] J. R. Cordy and C. K. Roy, "The NiCad clone detector," in *Proc. IEEE 19th Int. Conf. Program Comprehension*, Jun. 2011, pp. 219–220.
[38] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras, "The attack of the clones: A study of the impact of shared code on vulnerability patching," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 692–708.
[39] M. A. Bari and S. Ahamad, "Code cloning: The analysis, detection and removal," *Int. J. Comput. Appl.*, vol. 20, no. 7, pp. 34–38, Apr. 2011.
[40] D. Zhang, D. Liu, Y. Lei, D. Kung, C. Csallner, and W. Wang, "Detecting vulnerabilities in C programs using trace-based testing," in *Proc. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2010, pp. 241–250.
[41] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: Finding unpatched code clones in entire OS distributions," in *Proc. IEEE Symp. Secur. Privacy*, May 2012, pp. 48–62.
[42] H. Li, T. Kim, M. Bat-Erdene, and H. Lee, "Software vulnerability detection using backward trace analysis and symbolic execution," in *Proc. Int. Conf. Availability, Rel. Secur.*, Sep. 2013, pp. 446–454.
[43] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A scalable approach for vulnerable code clone discovery," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2017, pp. 595–614.
[44] C. Zhu, Y. Tang, Q. Wang, and M. Li, "Enhancing code similarity analysis for effective vulnerability detection," in *Proc. 2nd Int. Conf. Comput. Sci. Softw. Eng. (CSSE)*, 2019, pp. 153–158.
[45] B. Bowman and H. H. Huang, "VGRAPH: A robust vulnerable code clone detection system using code property triplets," in *Proc. 5th IEEE Eur. Symp. Secur. Privacy (EuroS P)*, Sep. 2020, pp. 53–69.
[46] S. Mishra and M. Polychronakis, "Saffire: Context-sensitive function specialization against code reuse attacks," in *Proc. 5th IEEE Eur. Symp. Secur. Privacy (EuroS P)*, Sep. 2020, pp. 17–33.
[47] S. Kim and H. Lee, "Software systems at risk: An empirical study of cloned vulnerabilities in practice," *Comput. Secur.*, vol. 77, pp. 720–736, Aug. 2018.
[48] K. Goseva-Popstojanova and A. Perhinschi, "On the capability of static code analysis to detect security vulnerabilities," *Inf. Softw. Technol.*, vol. 68, pp. 18–33, Dec. 2015.
[49] H. Choi, S. Zhu, and T. F. L. Porta, "SET: Detecting node clones in sensor networks," in *Proc. 3rd Int. Conf. Secur. Privacy Commun. Netw. Workshops (SecureComm)*, 2007, pp. 341–350.
[50] K. Xing, F. Liu, X. Cheng, and D. H. C. Du, "Real-time detection of clone attacks in wireless sensor networks," in *Proc. 28th Int. Conf. Distrib. Comput. Syst.*, Jun. 2008, pp. 3–10.
[51] B. Parno, A. Perrig, and V. Gligor, "Distributed detection of node replication attacks in sensor networks," in *Proc. IEEE Symp. Secur. Privacy (S P)*, May 2005, pp. 49–63.
[52] N. He, L. Wu, H. Wang, Y. Guo, and X. Jiang, "Characterizing code clones in the ethereum smart contract ecosystem," 2019, *arXiv:1905.00272*. [Online]. Available: http://arxiv.org/abs/1905.00272
[53] R. Tekchandani, R. Bhatia, and M. Singh, "Semantic code clone detection for Internet of Things applications using reaching definition and liveness analysis," *J. Supercomput.*, vol. 74, no. 9, p. 4199–4226, Sep. 2018, doi: 10.1007/s11227-016-1832-6.
[54] Z. Luo, B. Wang, Y. Tang, and W. Xie, "Semantic-based representation binary clone detection for cross-architectures in the Internet of Things," *Appl. Sci.*, vol. 9, no. 16, p. 3283, Aug. 2019.
[55] V. Sachidananda, S. Bhairav, and Y. Elovici, "OVER: Overhauling vulnerability detection for IoT through an adaptable and automated static analysis framework," in *Proc. 35th Annu. ACM Symp. Appl. Comput.*, Mar. 2020, pp. 729–738.
[56] S. Liu, M. Dibaei, Y. Tai, C. Chen, J. Zhang, and Y. Xiang, "Cyber vulnerability intelligence for Internet of Things binary," *IEEE Trans. Ind. Informat.*, vol. 16, no. 3, pp. 2154–2163, Mar. 2020.
[57] J. Shi, S. M. Kywe, and Y. Li, "Batch clone detection in RFID-enabled supply chain," in *Proc. IEEE Int. Conf. RFID (IEEE RFID)*, Apr. 2014, pp. 118–125.
[58] J. Huang, X. Li, C.-C. Xing, W. Wang, K. Hua, and S. Guo, "DTD: A novel double-track approach to clone detection for RFID-enabled supply chains," *IEEE Trans. Emerg. Topics Comput.*, vol. 5, no. 1, pp. 134–140, Jan. 2017.
[59] H. Maleki, R. Rahaeimehr, and M. van Dijk, "SoK: RFID-based clone detection mechanisms for supply chains," in *Proc. Workshop Attacks Solutions Hardw. Secur.*, Nov. 2017, pp. 33–41.
[60] H. Kamaludin, H. Mahdin, and J. H. Abawajy, "Clone tag detection in distributed RFID systems," *PLoS ONE*, vol. 13, no. 3, Mar. 2018, Art. no. e0193951.
[61] J. Bonneau, A. Miller, J. Clark, A. Narayanan, J. A. Kroll, and E. W. Felten, "SoK: Research perspectives and challenges for bitcoin and cryptocurrencies," in *Proc. IEEE Symp. Secur. Privacy*, May 2015, pp. 104–121.
[62] M. Conti, E. S. Kumar, C. Lal, and S. Ruj, "A survey on security and privacy issues of bitcoin," *IEEE Commun. Surveys Tuts.*, vol. 20, no. 4, pp. 3416–3452, May 2018.
[63] *Common Vulnerabilities and Exposures (CVE0)*, M. Corporation, USA, 2020.
[64] T. Unruh, B. Shastry, M. Skoruppa, F. Maggi, K. Rieck, J.-P. Seifert, and F. Yamaguchi, "Leveraging flawed tutorials for seeding large-scale Web vulnerability discovery," in *Proc. 11th USENIX Workshop Offensive Technol. (WOOT)*, 2017, pp. 1–12.
[65] K. Vineetha and N. S. Krishna, "Efficient code clone analysis to detect vulnerability in dynamic Web applications," *Int. J. Comput. Sci. Eng.*, vol. 4, no. 11, pp. 57–60, Nov. 2016.
[66] A. Agrawal, M. Alenezi, R. Kumar, and R. A. Khan, "A source code perspective framework to produce secure Web applications," *Comput. Fraud Secur.*, vol. 2019, no. 10, pp. 11–18, Oct. 2019.
[67] M. R. Farhadi, B. C. M. Fung, Y. B. Fung, P. Charland, S. Preda, and M. Debbabi, "Scalable code clone search for malware analysis," *Digit. Invest.*, vol. 15, pp. 46–60, Dec. 2015.
[68] M. H. Alalfi, E. P. Antony, and J. R. Cordy, "An approach to clone detection in sequence diagrams and its application to security analysis," *Softw. Syst. Model.*, vol. 17, no. 4, pp. 1287–1309, Oct. 2018.
[69] M. H. Alalfi, J. R. Cordy, and T. R. Dean, "Automated verification of role-based access control security models recovered from dynamic Web applications," in *Proc. 14th IEEE Int. Symp. Web Syst. Evol. (WSE)*, Sep. 2012, pp. 1–10.
[70] J. Akram, L. Qi, and P. Luo, "VCIPR: Vulnerable code is identifiable when a patch is released (Hacker's Perspective)," in *Proc. 12th IEEE Conf. Softw. Test., Validation Verification (ICST)*, Apr. 2019, pp. 402–413.

[71] H. Shi, R. Wang, Y. Fu, Y. Jiang, J. Dong, K. Tang, and J. Sun, "Vulnerable code clone detection for operating system through correlation-induced learning," *IEEE Trans. Ind. Informat.*, vol. 15, no. 12, pp. 6551–6559, Dec. 2019.
[72] S. Ducasse, O. Nierstrasz, and M. Rieger, "On the effectiveness of clone detection by string matching," *J. Softw. Maintenance Evol., Res. Pract.*, vol. 18, no. 1, pp. 37–58, 2006.
[73] J. H. Johnson, "Substring matching for clone detection and change tracking," in *Proc. ICSM*, vol. 94, 1994, pp. 120–126.
[74] A. Marcus and J. I. Maletic, "Identification of high-level concept clones in source code," in *Proc. 16th Annu. Int. Conf. Automated Softw. Eng. (ASE)*, 2001, pp. 107–114.
[75] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: A tool for finding copy-paste and related bugs in operating system code," in *Proc. OSDI*, 2004, vol. 4, no. 19, pp. 289–302.
[76] M. R. Farhadi, "Assembly code clone detection for malware binaries," Ph.D. dissertation, School Eng. Comput. Sci., Concordia Univ., Montreal, QC, Canada, 2013.
[77] I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Proc. Int. Conf. Softw. Maintenance*, 1998, pp. 368–377.
[78] V. Wahler, D. Seipel, J. Wolff, and G. Fischer, "Clone detection in source code by frequent itemset techniques," in *Proc. 4th IEEE Int. Workshop Source Code Anal. Manipulation*, Sep. 2004, pp. 128–135.
[79] K. A. Kontogiannis, R. Demori, E. Merlo, M. Galler, and M. Bernstein, "Pattern matching for clone and concept detection," *Automated Softw. Eng.*, vol. 3, nos. 1–2, pp. 77–108, Jun. 1996.
[80] J. Mayrand, C. Leblanc, and E. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Proc. ICSM*, 1996, p. 244.
[81] W. M. Khoo, A. Mycroft, and R. Anderson, "Rendezvous: A search engine for binary code," in *Proc. 10th Work. Conf. Mining Softw. Repositories (MSR)*, May 2013, pp. 329–338.
[82] Y. Hu, Y. Zhang, J. Li, and D. Gu, "Binary code clone detection across architectures and compiling configurations," in *Proc. IEEE/ACM 25th Int. Conf. Program Comprehension (ICPC)*, May 2017, pp. 88–98.
[83] Y. J. Lee, S.-H. Choi, C. Kim, S.-H. Lim, and K.-W. Park, "Learning binary code with deep learning to detect software weakness," in *Proc. KSII 9th Int. Conf. Internet (ICONI) Symp.*, 2017, pp. 1–5.
[84] A. Sæbjørnsen, J. Willcock, T. Panas, D. Quinlan, and Z. Su, "Detecting code clones in binary executables," in *Proc. 18th Int. Symp. Softw. Test. Anal. (ISSTA)*, 2009, pp. 117–128.
[85] H. Xue, G. Venkataramani, and T. Lan, "Clone-slicer: Detecting domain specific binary code clones through program slicing," in *Proc. Workshop Forming Ecosystem Around Softw. Transformation (FEAST)*, 2018, pp. 27–33.
[86] H. Xue, G. Venkataramani, and T. Lan, "Clone-hunter: Accelerated bound checks elimination via binary code clone detection," in *Proc. 2nd ACM SIGPLAN Int. Workshop Mach. Learn. Program. Lang.*, Jun. 2018, pp. 11–19.
[87] Y. A. N. Ishiura, "Detection of vulnerability guard elimination by compiler optimization based on binary code comparison," in *Proc. 22nd Workshop Synth. Syst. Integr. Mixed Inf. Technol.*, Japan, 2019.
[88] S. H. H. Ding, B. C. M. Fung, and P. Charland, "Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *Proc. IEEE Symp. Secur. Privacy (SP)*, May 2019, pp. 472–489.
[89] G. Erdélyi and E. V. Carrera, "Digital genome mapping: Advanced binary malware analysis," in *Proc. Virus Bull. Conf.*, Chicago, IL, USA, 2004.
[90] S. Mishra and M. Polychronakis, "Shredder: Breaking exploits through API specialization," in *Proc. 34th Annu. Comput. Secur. Appl. Conf.*, Dec. 2018, pp. 1–16.
[91] J. Akram, Z. Shi, M. Mumtaz, and P. Luo, "DCCD: An efficient and scalable distributed code clone detection technique for big code," in *Proc. SEKE*, Jul. 2018, pp. 353–354.
[92] R. Russell, L. Kim, L. Hamilton, T. Lazovich, J. Harer, O. Ozdemir, P. Ellingwood, and M. McConley, "Automated vulnerability detection in source code using deep representation learning," in *Proc. 17th IEEE Int. Conf. Mach. Learn. Appl. (ICMLA)*, Dec. 2018, pp. 757–762.
[93] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, E. Antelman, A. Mackay, M. W. McConley, J. M. Opper, P. Chin, and T. Lazovich, "Automated software vulnerability detection with machine learning," 2018, *arXiv:1803.04497*. [Online]. Available: http://arxiv.org/abs/1803.04497
[94] S. Kim, S. Hong, J. Oh, and H. Lee, "Obfuscated VBA macro detection using machine learning," in *Proc. 48th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2018, pp. 490–501.
[95] X. Wang, K. Sun, A. Batcheller, and S. Jajodia, "Detecting, '0-day' vulnerability: An empirical study of secret security patch in OSS," in *Proc. 49th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw. (DSN)*, Jun. 2019, pp. 485–492.
[96] 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 *Proc. 32nd Annu. Conf. Comput. Secur. Appl.*, Dec. 2016, pp. 201–213.
[97] Z. Li, D. Zou, S. Xu, H. Jin, Y. Zhu, and Z. Chen, "SySeVR: A framework for using deep learning to detect software vulnerabilities," 2018, *arXiv:1807.06756*. [Online]. Available: http://arxiv.org/abs/1807.06756
[98] F. Ullah, M. R. Naeem, L. Mostarda, and S. A. Shah, "Clone detection in 5G-enabled social IoT system using graph semantics and deep learning model," *Int. J. Mach. Learn. Cybern.*, vol. 12, pp. 1–13, Jan. 2021.
[99] P.-Y. Lee, C.-M. Yu, T. Dargahi, M. Conti, and G. Bianchi, "MDSClone: Multidimensional scaling aided clone detection in Internet of Things," *IEEE Trans. Inf. Forensics Security*, vol. 13, no. 8, pp. 2031–2046, Aug. 2018.



HAIBO ZHANG received the B.S. degree in software engineering from Anhui University, China, in 2015, and the M.S. degree in cyber security engineering from the Viterbi School of Engineering, University of Southern California, USA, in 2018. She is currently pursuing the Ph.D. degree in cyber security with Kyushu University, Japan. She is also working as a Research Assistant with Strategic International Collaborative Research Program (SICORP) for Internet of Things related research work. Her research interests include Internet of smart things security, digital supply chain security, and vulnerable software clone detection.

KOUICHI SAKURAI (Member, IEEE) received the B.S. degree in mathematics from the Faculty of Science, Kyushu University, in 1986, the M.S. degree in applied science and the Ph.D. degree in engineering from the Faculty of Engineering, Kyushu University, in 1988 and 1993, respectively. From 1988 to 1994, he was engaged in research and development on cryptography and information security at the Computer and Information Systems Laboratory, Mitsubishi Electric Corporation. Since 1994, he has been working with the Department of Computer Science, Kyushu University, as an Associate Professor, where he became a Full Professor, in 2002. He is concurrently working with the Institute of Systems, Information Technologies and Nanotechnologies, as the Chief of the Information Security Laboratory, for promoting research co-operations among the industry, university, and government under the theme Enhancing IT-Security in Social Systems. From 2005 to 2006, he was successful in generating such co-operation between Japan, China, and Korea, for security technologies, as a Leader of the Cooperative International Research Project supported by the National Institute of Information and Communications Technology (NICT). Moreover, in March 2006, he established research co-operations under a Memorandum of Understanding in the field of information security with Prof. Bimal Kumar Roy, the first time Japan has partnered with The Cryptology Research Society of India (CRSI). He currently directs the Laboratory for Information Technology and Multimedia Security and is working with the CyberSecurity Center, Kyushu University. He is also with Department of Advanced Security, Advanced Telecommunications Research Institute International and involved in a NEDO-SIP-Project on supply chain security. He has published about 400 academic articles in cryptography and cybersecurity.