Available online at www.sciencedirect.com

ScienceDirect

journal homepage: www.elsevier.com/locate/cose

Computers & Security

Software systems at risk: An empirical study of cloned vulnerabilities in practice

Seulbae Kim, Heejo Lee[*]

Department of Computer Science and Engineering, Korea University, Seoul 136-713, Republic of Korea

ARTICLE INFO

Article history:
Received 1 October 2017
Revised 10 February 2018
Accepted 12 February 2018
Available online 27 February 2018

Keywords:
Software vulnerability
Open source software
Vulnerability propagation
Code clone detection
Vulnerability exploitation

ABSTRACT

With the growth of open source software (OSS), code clones – code fragments that are copied and pasted within or between software systems – are proliferating. Although code cloning may expedite the process of software development, it often critically affects the security of software because vulnerabilities and bugs can easily be propagated through code clones. These vulnerable code clones are increasing in conjunction with the growth of OSS, potentially contaminating many systems. Although researchers have attempted to detect code clones for decades, most of these attempts fail to scale to the size of the ever-growing OSS code base. The lack of scalability prevents software developers from readily managing code clones and associated vulnerabilities. Moreover, most existing clone detection techniques focus overly on merely detecting clones and this impairs their ability to accurately find “vulnerable” clones.

In this paper, we propose VUDDY, an approach for the scalable detection of vulnerable code clones, which is capable of detecting security vulnerabilities in large software programs efficiently and accurately. Its extreme scalability is achieved by leveraging function-level granularity and a length-filtering technique that reduces the number of signature comparisons. This efficient design enables VUDDY to preprocess a billion lines of code in 14 hours and 17 minutes, after which it requires a few seconds to identify code clones. In addition, we designed a vulnerability-preserving abstraction technique that renders VUDDY resilient to common modifications in cloned code, while preserving the vulnerable conditions even after the abstraction is applied. This extends the scope of VUDDY to identifying variants of known vulnerabilities, with high accuracy. An implementation of VUDDY has been serviced online for free at IoTcube, an automated vulnerability detection platform. In this study, we describe its principles, evaluate its efficacy, and analyze the vulnerabilities VUDDY detected in various real-world software systems, such as Apache HTTPD server and an Android smartphone.

1. Introduction

For the past decade, open source software (OSS) has grown rapidly in size, and is becoming a foundation for a majority of applications, operating systems, databases, and web servers. The number of repositories in GitHub increased from 1 million to 10 million between July 2010 and December 2013, and then to more than 66 million in August 2017, with most of the repositories being software projects (GitHub). Black Duck Software and North Bridge revealed in 2016 Future of Open Source survey that 78% of the companies run open source

* Corresponding author.
E-mail address: heejo@korea.ac.kr (H. Lee).

https://doi.org/10.1016/j.cose.2018.02.007
0167-4048/© 2018 Elsevier Ltd. All rights reserved.

software (Blackduck). In literature, researchers showed that open source projects have linear to quadratic growth patterns (Godfrey and Tu, 2000; Scacchi, 2006; Succi et al., 2001).

The considerable increase and the ubiquitous use of OSS programs have naturally led to an increase in software vulnerabilities caused by code cloning, thereby posing dire threats to the security of software systems (Li et al., 2015). Numerous instances in practice substantiate the claim that software vulnerabilities are propagated through code cloning (Dang et al., 2017). For example, the OpenSSL Heartbleed vulnerability (CVE-2014-0160) has affected a number of different types of systems including web applications as well as OS distributions, and even hardware products such as routers and switches, because the affected systems cloned a part or the entire OpenSSL library into their systems.

Moreover, the life cycle of vulnerabilities even aggravates such problem. That is, even if a vendor were to release a security patch immediately after the discovery of vulnerability in the original program, it would take time for the patch to be fully deployed through every program that cloned the vulnerable code of the original program (Nappa et al., 2015). This “time lag” between patch release and deployment increases when more steps and parties are involved in manufacturing an end product. For example, Google takes the Linux kernel to make the Android operating system, and then smartphone vendors take the Android OS in order to make firmware for their smartphone. Therefore, we can hardly expect that security patches issued for Linux kernel are immediately applied to smartphone’s firmware. Taking advantage of this fact, the attackers in 2016 could successfully deploy the “Dogspectus” ransomware to a lot of Android smartphones by exploiting two-year-old kernel vulnerability (CVE-2014-3153) residing in the firmware.

To address such clone-related problems, many researchers have proposed code clone detection techniques (Rattan et al., 2013). However, few techniques are suitable for accurately finding vulnerability in a scalable manner. Token-based lexical techniques such as CCFinder (Kamiya et al., 2002) and CP-Miner (Li et al., 2006) not only suffer from low scalability of complex token sequence comparing algorithms they take, but also have high false positive rate caused by their aggressive abstraction and filtering heuristics. This implies that this design does not guarantee sufficient reliability to be useful for vulnerability detection. Similarly, abstract data structure based approaches (e.g., Deckard (Jiang et al., 2007b), or Baxter’s (Baxter et al., 1998)) have to apply expensive tree-matching operations or graph mining techniques for similarity estimation. Although such approaches would be capable of discovering code fragments with similar syntactic patterns, this does not guarantee an accurate vulnerability detection because two code fragments with identical abstract syntax trees (ASTs) do not necessarily contain the same vulnerability (Jiang et al., 2007a). SourcererCC (Sajnani et al., 2016) uses a bag-of-token strategy to manage minor to specific changes in clones, which impairs the accuracy and results in high false positive rate. Deep learning approach was also proposed (White et al., 2016), but its effectiveness in large-scale code bases is not verified. Notable exception is ReDeBug (Jang et al., 2012) which aims to achieve both accuracy and scalability by applying hash functions to lines of code and later detecting clones by comparing hash values through bloom filter. However, when it comes to finding vulnerable code clones in massive code bases, ReDeBug is still not satisfactory both in terms of accuracy and scalability. Some techniques leverage a combination of various approaches. VulPecker (Li et al., 2016) characterizes a vulnerability with a predefined set of features, then selects one of the existing code-similarity algorithms (e.g. Jang et al. (2012), Li et al. (2006) and Pham et al. (2010) which is optimal for the type of vulnerable code fragment. A considerable amount of time required for the learning phase and algorithm selection makes it improper to be used against massive open source projects. An approach to analyze code heterogeneity for detecting repackaged Android malware (Tian et al., 2016, 2017) utilizes machine learning algorithms, which are heavily dependent on the manual efforts to select features.

Meanwhile, our preliminary work, called VUDDY (Kim et al., 2017a), proposed the most scalable and accurate approach for vulnerable code clone detection by leveraging function-level granularity and applying vulnerability-preserving abstraction method. In this paper, we briefly describe the approach we used for scalable and accurate detection of vulnerable code clones, then provide a detailed analysis of code reuse patterns and associated vulnerabilities in real-world software systems. In practice, we discovered several kernel vulnerabilities in the recently released smartphones, and successfully exploited the vulnerabilities. In addition, we identified several famous open source software which are released with old, vulnerable libraries. These results show that security patches are deployed at a slow pace if code is reused, and because of this, unpatched vulnerabilities can easily be exploited for malicious activities even though they are not zero-day vulnerabilities.

The contributions of this study include:

- *Scalable clone detection:* We propose VUDDY, an approach to scalable yet accurate code clone detection, which adopts a robust parsing and a novel fingerprinting mechanism for functions. VUDDY is able to process a billion lines of code in only 14 hours and 17 minutes, which is an unprecedented speed.
- *Vulnerability-preserving abstraction:* We present an effective abstraction scheme optimized for detecting unknown vulnerable code clones. This allows VUDDY to detect unknown vulnerable code clones, as well as known vulnerabilities in a target program. Owing to this design, VUDDY detects 24% more vulnerable clones which are unknown variants of known vulnerabilities.
- *Analysis of cloned vulnerabilities in real-world software systems:* We analyze and verify the cloned vulnerabilities detected in the software systems in practice, including famous projects on GitHub and SourceForge, and smartphones with high market shares. Our analysis draws a significant insight that there is a vicious cycle of vulnerable code clone propagation and prolonged patch distribution.
- *Open service:* We have been servicing VUDDY as a form of open web service at no charge, since April 2016. In practice, VUDDY is being used by many in the open source community and by IoT device manufacturers, for the purpose of examining their software. In the past 17 months, 20 billion lines of code have been queried to our open service, and

253,036 vulnerable functions have been detected. Please see https://iotcube.net/.

2. Vulnerable code clone discovery

In this section, we describe our preliminary work VUDDY (VUlnerable coDe clone DiscoverY), which is a scalable approach to code clone detection that can be seamlessly applied to the massive OSS pool.

2.1. *Goals and scope*

The types of code clones have to be clarified in order to address the goals and scope of VUDDY. According to relevant previous research (Bellon et al., 2007; Koschke, 2007; Rattan et al., 2013; Roy et al., 2009), code clones can be categorized into the following 4 types:

- **Type-1: Exact clones.** Code fragments are duplicated as is, and completely identical.
- **Type-2: Renamed clones.** Code fragments are syntactically identical, but include modification of types, identifiers, comments, and whitespace.
- **Type-3: Restructured clones.** The structure of the cloned code fragment is modified (e.g., removal, insertion, or rearrangement of statements). For example, if an unnecessary statement that declares an unused variable is removed from a cloned code, it would be a Type-3 code clone.
- **Type-4: Semantic clones.** Code fragments are functionally identical, but are syntactically different.

VUDDY is devised for detecting code clones which are security vulnerabilities. Specifically, its goal is to promptly and accurately discover the code clones between a corpus of vulnerable code and a target program. To achieve that goal, we designed VUDDY to be able to detect *Type-1* and *Type-2* clones because of the following three reasons:

1. *Reducing false negatives*: By detecting Type-1 and Type-2 clones, VUDDY becomes resilient to minor code changes that frequently occur. In other words, VUDDY is able to detect exact clones of vulnerability, as well as clones in which variable names, identifiers, data types, comments, and whitespace are modified. Namely, VUDDY can prevent any vulnerability that is in the database (i.e., the collection of vulnerable code), and the variants of these vulnerabilities. This is further addressed in subsection 2.4, and the examples of such vulnerable clones can be found in section 4.
2. *Reducing false positives*: Type-3 and Type-4 code clones are deliberately excluded from the scope of VUDDY, because they are subjective to the loss of syntactic information which are crucial for a vulnerability to be triggered. We can observe numerous cases where security vulnerabilities are very sensitive to the order of statements and constants. For example, a severe vulnerability that allows remote code execution attacks from the attackers can be patched by adding one sentence that sanitizes an input. In case of CVE-2012-0876, a hash DoS vulnerability found in Expat XML parsing library, the problematic function was patched by changing a constant value 0 to a salt variable. VUDDY can distinguish unpatched (vulnerable) code from patched code.
3. *Increasing scalability*: Analyzing semantics for determining the equivalence of code is time-consuming and error-prone. However, a typical parsing is enough for the recognition of the syntax and the detection of Type-1 and Type-2 clones, which makes VUDDY lightweight and fast.

2.2. *Code clone detector*

Fig. 1 illustrates the overall stages and substeps of VUDDY. VUDDY preprocesses a target program and generates a fingerprint dictionary. And then, it detects code clones by comparing two or more fingerprint dictionaries. By generating a fingerprint dictionary consisting of vulnerable functions, and comparing that dictionary with a dictionary generated from target program, VUDDY will disclose vulnerable code clones in the target program.

2.2.1. *Preprocessing*

S1: Function retrieval. The Preprocessing stage begins by retrieving functions from a given program by using a robust parser. VUDDY then performs a syntax analysis to identify formal parameters, data types in use, local variables, and function calls. This supplementary information is used in the next stage: abstraction and normalization.

S2: Abstraction and normalization. In this stage, an abstraction and normalization feature is offered. To make VUDDY resilient to common code modifications while preserving vulnerable condition, every occurrence of formal parameters, local variables, data types, and function calls that appear in the body of a function is replaced with symbols FPARAM, LVAR, DTYPE, and FUNCCALL, respectively. The body is then normalized by removing every comment, whitespace, tab, and line feed character, and by converting all characters into lowercase. After applying abstraction and normalization, VUDDY is not affected by any syntactically meaningless modifications (e.g., function inlining, or adding comments). Moreover, VUDDY can capture changes in function calls and

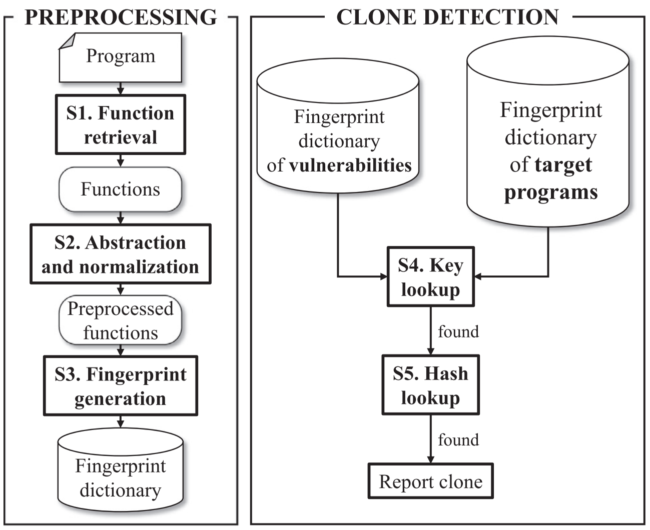


Fig. 1 – Two stages of VUDDY: preprocessing and clone detection.

```
Level 0:  No abstraction.
void avg (float arr[], int len) {
  static float sum = 0;
  unsigned int i;
  for (i = 0; i < len; i++);
    sum += arr[i];
  printf("%f %d",sum/len,validate(sum));
}
Level 1:  Formal parameter abstraction.
void avg (float FPARAM[], int FPARAM) {
  static float sum = 0;
  unsigned int i;
  for (i = 0; i < FPARAM; i++)
    sum += FPARAM[i];
  printf("%f %d",sum/FPARAM,validate(sum);
}
Level 2:  Local variable name abstraction.
void avg (float FPARAM[], int FPARAM) {
  static float LVAR = 0;
  unsigned int LVAR;
  for (LVAR = 0; LVAR < FPARAM; LVAR++)
    LVAR += FPARAM[LVAR];
  printf("%f %d",LVAR/FPARAM,validate(LVAR));
}
Level 3:  Data type abstraction.
void avg (float FPARAM[], int FPARAM) {
  DTYPE LVAR = 0;
  unsigned DTYPE LVAR;
  for (LVAR = 0; LVAR < FPARAM; LVAR++)
    LVAR += FPARAM[LVAR];
  printf("%f %d",LVAR/FPARAM,validate(LVAR));
}
Level 4:  Function call abstraction.
void avg (float FPARAM[], int FPARAM) {
  DTYPE LVAR = 0;
  unsigned DTYPE LVAR;
  for (LVAR = 0; LVAR < FPARAM; LVAR)
    LVAR += FPARAM[LVAR];
  FUNCCALL("%f %d",LVAR/FPARAM,FUNCCALL(LVAR));
}
```

Fig. 2 – Level-by-level application of abstraction schemes on a sample function.

shared APIs, which are typical causes of recurring vulnerabilities (Yamaguchi et al., 2012; Zhang et al., 2014). Fig. 2 shows the transformation of a sample function at varying abstraction levels. Here, higher levels of abstraction include subordinate levels.

S3: Fingerprint generation. VUDDY generates fingerprints for the retrieved function bodies that are abstracted and normalized. A fingerprint of a function is represented as a 2-tuple where the length of the normalized function body string becomes one element, and the hash value of the string becomes the other. Fig. 3 shows the fingerprinting of example functions.

After fingerprinting, VUDDY stores the tuples in a dictionary that maps keys to values, where the length values (i.e., the first element of a tuple) are keys, and the hash values that share the same key are mapped to each key. Fig. 4 shows how the example functions of Fig. 3 are classified and stored in a dictionary.

```
Original:       int sum (int a, int b) {
                  return a + b;
                }
Preprocessed:   returnfparam+fparam;
Length:         20
Hash value:     c94d99100e084297ddbf383830f655d1
Fingerprint:    {20, c94d99100e084297ddbf383830f655d1}
Original:       void increment () {
                  int num = 80;
                  num++; /* no return val */
                }
Preprocessed:   dtypelvar=80;lvar++;
Length:         20
Hash value:     d6e77882a5c55c67f45f5fd84e1d616b
Fingerprint:    {20, d6e77882a5c55c67f45f5fd84e1d616b}
Original:       void printer (char* src) {
                  printf("%s", src);
                }
Preprocessed:   funccall("%s",fparam);
Length:         23
Hash value:     9a45e4a15c928699afe867e97fe839d0
Fingerprint:    {23, 9a45e4a15c928699afe867e97fe839d0}
```

Fig. 3 – Example functions and corresponding fingerprints.

In the dictionary shown in Fig. 4, the two functions in Fig. 3 (sum and increment) are classified under the same integer key, because the length of their abstracted and normalized bodies is identical as 20. The fingerprint of the other function (printer) is assigned to another key, 23, in the dictionary. In practice, we ignore functions of which the lengths are shorter than 50, to prevent VUDDY from identifying short functions as clones. Intuitively, short functions are hardly vulnerable by themselves.

2.2.2. *Clone detection*

After two identical functions are preprocessed, they are required to have the same lengths even if variables are renamed and comments are changed. Leveraging that fact, VUDDY detects code clones between two programs by performing at most two membership tests for each length-classified fingerprint dictionary: a key lookup, and a subsequent hash lookup.

S4: Key lookup. VUDDY performs the first membership testing, by iterating over every key in a source dictionary, and looking for the existence of the key (i.e., the length of the preprocessed function) in the target fingerprint dictionary. If the key lookup fails, then VUDDY concludes that there is no clone in the target program. If it succeeds to find the existence of the

```
20:  {
       'c94d99100e084297ddbf383830f655d1',
       'd6e77882a5c55c67f45f5fd84e1d616b'
}
23:  {
       '9a45e4a15c928699afe867e97fe839d0'
}
```

Fig. 4 – A dictionary that stores the fingerprints of the example functions. A set containing two hash values is mapped to the key 20, which is the length value, and another set is mapped to the key 23.

same integer key, then VUDDY proceeds to the next substep: Hash lookup.

S5: Hash lookup. As a last substep of clone detection, VUDDY searches for the presence of the hash value in the set mapped to the integer key. If the hash value is discovered, then the function is considered to be a clone. For example, when comparing dictionary A and B, VUDDY iterates S4 over every key in dictionary A, searching for the key in dictionary B. For each key shared by dictionary A and B, VUDDY performs S5 to retrieve all shared hash values, which are the clones we are looking for.

The design of VUDDY accelerates the process of clone searching by taking advantage of the following three facts:

a. The time complexity of an operation that checks the existence of a value from a set of unique elements is $O(1)$ on average, and $O(n)$ in the worst case, where n is the number of elements in a set.
b. It is guaranteed that even in the worst case, n is small because of the length classification. For example, the fingerprint dictionary of Linux kernel 4.7.6 (23 K files with over 15.4 MLoC) only contains 5245 integer keys, and among the hash sets associated to the keys, the largest set has 1019 elements. The average number of elements of the hash sets is 67.85, the median is 5, and the mode (the value that occurs most often) is 1. This implies that most of the hash set will have only one element.
c. Once the preprocessing is complete, the resulting fingerprint dictionary can be permanently reused, unless some portion of the program is changed. This efficient design enables VUDDY to perform a real-time clone detection.

2.3. *Establishing a vulnerability database*

A reliable dictionary of vulnerable functions is a prerequisite for accurate vulnerable code clone detection. To obtain various vulnerable functions, we leveraged the Git repositories of well-known authoritative open source projects, and Subversion (SVN) repositories of widely-used software libraries. The repositories are shown in Table 1.

The process of collecting vulnerable code and establishing a vulnerability database is fully automated, which can be considered as the process of reverse patching: from security patches we reconstruct the unpatched, vulnerable function. The process of reconstructing vulnerable functions from Git or SVN repositories consists of the following steps:

a. **`git clone` repository.** This is to download specified git repository into a local directory. For SVN repositories, use "`git svn clone` repository" command instead, which clones an SVN repository and then converts it into a Git repository.
b. **`git log --grep=''CVE-20''` for each repository.** This searches for the commits regarding Common Vulnerability and Exposures (CVEs). Other general keywords such as "buffer overflow" or "heartbleed vulnerability" instead of "CVE-20" work as well.
c. **`git show` the searched commits.** This command shows the full commit log which contains a description of the vulnerability, as well as a security patch in unified diff format.

Table 1 – Number of CVEs automatically collected from each Git or SVN repositories.

Type	Repository Name	# CVEs
Operating system	Codeaurora Android	1966
	OpenSUSE kernel	701
	Google Android	418
	Ubuntu-Trusty	324
	FreeBSD	244
	Linux kernel	185
Database server	PostgreSQL	55
	MySQL	1
Web server & browser	Google Chromium	1145
	ChakraCore	62
	Apache HTTPD	57
	Gecko	6
	Nginx	2
Library, protocol, language, others	OpenSSL	117
	Kerberos5	72
	PHP	36
	GlibC	35
	BotanTLS	10
	389 Directory Server	9
	PCRE	5
	Wireshark	3

Every patch includes reference IDs to the old and new files addressed by the patch.

d. **Filter irrelevant commits.** The steps listed could fetch commits that are inappropriate for vulnerability detection. For example, some commits have the keyword "CVE-20" in their message, which is actually "Revert the patch for CVE-20XX-XXXX." Merging commits or updating commits which usually puts all the messages of associated commits together is another problem, particularly if one of the commits happens to be a CVE patch. In such cases, our automated approach would end up retrieving a benign function. Thus, commits which revert, merge, or update are discarded in this step.
e. **`git show` the old file ID and retrieve vulnerable function.** This shows the old, thus unpatched version of the file. We then retrieve the vulnerable function from the file.

Listing 1: Patch for CVE-2013-4312.

```
1 diff --git a/fs/pipe.c b/fs/pipe.c
2 index d2cbeff..19078bd 100644
3 --- a/fs/pipe.c
4 +++ b/fs/pipe.c
5 @@ -607,6 +642,8 @@ void free_pipe_info(struct
      pipe_inode_info *pipe)
6 {
7 int i;
8
9 + account_pipe_buffers(pipe, pipe->buffers, 0);
10 + free_uid(pipe->user);
11 for (i = 0; i < pipe->buffers; i++) {
12 struct pipe_buffer *buf = pipe->bufs + i;
13 if (buf->ops)
```

Listing 1 is the patch for CVE-2013-4312, found in the Codeaurora Android repository. This patch adds lines 9 and 10 to ensure that the per-user amount of pages allocated in pipes is limited so that the system can be protected against memory abuse. The file metadata in line 2 indicates the references to the old file (`d2cbeff`) and the new file (`19078bd`), and line 5 conveys information about the line numbers of the affected portion in the file.

We could retrieve the old function, namely the vulnerable version of the function, by querying "`git show d2cbeff`" to the cloned Git object, obtaining the old file, and parsing the relevant function. Listing 2 is the retrieved vulnerable function, which includes both the vulnerable part, and the context around it.

Listing 2: Vulnerable function retrieved from the patch for CVE-2013-4312.

```
void free_pipe_info(struct pipe_inode_info *pipe)
{
    int i;

    for (i = 0; i < pipe->buffers; i++) {
        struct pipe_buffer *buf = pipe->bufs + i;
        if (buf->ops)
            buf->ops->release(pipe, buf);
...
```

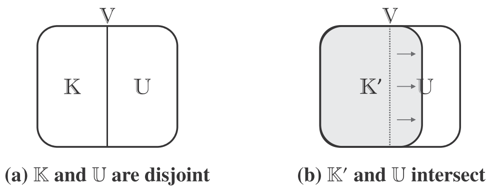
Our automated system collected 11,146 vulnerable functions which correspond to 3551 unique CVEs and 38 CWEs. These vulnerable functions include various types of vulnerability, such as buffer overflow, integer overflow, input validation error, permission-related vulnerabilities, and others.

2.4. *Vulnerable code clone detection*

The application of VUDDY for vulnerability detection does not require any supplementary procedure. VUDDY processes the functions in the vulnerability database in the same way as it does with a normal program, then discovers vulnerability in the target program by detecting code clones between the vulnerability database and the target program.

Here, we can determine which vulnerability VUDDY is capable of discovering. As illustrated in Fig. 5(a), if set $\mathbb{K}$ is the set of every known vulnerability, then $\mathbb{K} \subset \mathbb{V}$ where $\mathbb{V}$ is the set consisting of all vulnerabilities. Naturally, we can regard $\mathbb{U}$, the set of unknown vulnerabilities, and $\mathbb{K}$ as being disjoint, so that $\mathbb{K} \cup \mathbb{U} = \mathbb{V}$ and $\mathbb{K} \cap \mathbb{U} = \phi$. If a clone detector only considers exact clones (i.e., code fragments that are duplicated without any change), then the coverage is $\mathbb{K}$. However, by the use of our abstraction strategy, the coverage of an abstract clone detector can also cover vulnerabilities in $\mathbb{K}'$ as depicted in Fig. 5(b), which is a set of abstract vulnerabilities. This means that VUDDY can detect known vulnerabilities, as well as variants of the known vulnerabilities, which are in $\mathbb{K}'$, where $|\mathbb{K}' \cap \mathbb{U}| > 0$. $\mathbb{K}' \cap \mathbb{U}$ is the set of unknown vulnerable code clones discovered by VUDDY.

The source code and implementation of VUDDY are available online at https://github.com/squizz617/vuddy.



V
K
U
V
K'
U
(a) $\mathbb{K}$ and $\mathbb{U}$ are disjoint
(b) $\mathbb{K}'$ and $\mathbb{U}$ intersect

Fig. 5 – Relationship between known, unknown and variants of known vulnerabilities.

3. Theoretical evaluation

In this section we evaluate the scalability and accuracy of VUDDY, by comparing VUDDY against four publicly available and competitive techniques: SourcererCC, ReDeBug, Deckard, and CCFinderX.

3.1. *Experimental setup and dataset*

System environment: We evaluated the execution and detection performance of VUDDY by conducting experiments on a machine running Ubuntu 16.04, with a 2.40 GHz Intel Zeon processor, 32 GB RAM, and 6 TB HDD.

Dataset: We collected our target C/C++ programs from GitHub. These programs had at least one star and were pushed at least once during the period from January 1, 2016 to July 28, 2016. Repositories that are starred (i.e., bookmarked by GitHub users) are popular and influential repositories. The existence of a push record during the first half of 2016 implies that the repository is active. The repository cloning process required 7 weeks to finish, gathering 25,253 Git repositories which satisfy the aforementioned two conditions. In addition to the GitHub projects, we downloaded the firmware of several Android smartphones from the web pages of the manufacturers.

Configuration: The configuration choices can have a significant impact on the behavior of the tools that are compared (Wang et al., 2013). As a remedy, we referenced the optimal configuration of each technique found by previous studies (Svajlenko and Roy, 2014, 2015; Wang et al., 2013; Sajnani et al., 2016) to conduct a sufficiently fair evaluation.

3.2. *Scalability evaluation*

To measure the scalability of tools when handling real-world programs, we generated target sets of varying sizes, from 1 KLoC to 1 BLoC, by randomly selecting projects from the 25,253 Git projects we collected. All experiments were iterated five times each (except for SourcererCC, with which we iterated twice), to ensure that the results are reliable.

As described in Table 2, VUDDY (using function-level granularity) overwhelmed other techniques. Excluding Deckard, the results accord perfectly with an intuition that finer granularity leads to bad scalability. Deckard (using AST as granularity) had the least scalability, failing to process 100 MLoC target because of a memory error. Its low scalability can be attributed to the fundamental limitation of subgraph isomorphism problem, which is heavy and time-consuming. In the case of CCFinderX (using token-level granularity), a file I/O error occurred after 3 days of execution for a 1 BLoC target. Token is the smallest unit, meanwhile, VUDDY finished generating fingerprints and detecting clones of the 1 BLoC target in only 14 hours and 17 minutes. Although SourcererCC (using bag-of-tokens as granularity) and ReDeBug (using lines as granularity) also scaled to 1 BLoC, their execution is considerably slower than that of VUDDY. ReDeBug required more than a day, and SourcererCC required 25 days to finish detecting clones from the same 1 BLoC target. In fact, owing to the function-level granularity and the efficient matching algorithm, VUDDY

Table 2 – Scalability and time comparison for varying input size. The average time was computed after iterating five times for each experiment. Units in the parentheses indicate the granularity of each tool.

Input LoC	VUDDY (Function)	ReDeBug (Lines)	SourcererCC (Bag-of-words)	CCFinderX (Tokens)	Deckard (AST)
1 K	0.44 s	35.6 s	2.3 s	6 s	1 s
10 K	0.81 s	35.6 s	3.1 s	10 s	3 s
100 K	5.17 s	42 s	50.7 s	50 s	13 s
1 M	55 s	1 m 43 s	1 m 44 s	6 m 44 s	2 m 20 s
10 M	12 m 43 s	18 m 32 s	24 m 38 s	1 h 36 m	12h 30 m
100 M	1 h 32 m	2 h 32 m	9 h 42 m	12 h 44 m	Memory ERROR
1 B	14 h 17 m	1 d 3 h	25 d 3 h	File I/O ERROR	–

scales even to the size of all 25,253 repositories consisting of 8.7 BLoC with ease, requiring only 4 days and 7 hours.

In addition, we conducted an in-depth comparison of VUDDY with ReDeBug, which is the most scalable approach among the four techniques that are compared. When querying 10,469 vulnerable functions targeting an Android firmware (14.9 MLoC, using kernel version 3.18.14), VUDDY required 1024 seconds, while ReDeBug required 1695 seconds. In fact, VUDDY required 1023 seconds (99.9%) for preprocessing and for the fingerprint generation procedure, and the actual code clone detection required only 1.04 seconds, as illustrated in Fig. 6. Note that once the preprocessing is complete, VUDDY does not need to regenerate the fingerprint dictionary for every clone detection. This is not the case for ReDeBug. When executed, ReDeBug required 676 seconds and 1019 seconds for preprocessing and clone detection, respectively.

Fig. 7 displays a graph depicting the clone detection time of VUDDY and ReDeBug when varying size of target programs (1 KLoC to 1 BLoC) were given. The clone detection time of VUDDY is near-constant (approximately 1 second), while that of ReDeBug grows linearly with the size of target. Thus, we can conclude that VUDDY detects vulnerable code clones at a speed more than 10 times faster than ReDeBug, in practice, and this gap increases as target size grows.

3.3. *Accuracy evaluation*

Now we evaluate the accuracy of VUDDY by comparing the number of false positives produced by each tool, given a set of vulnerabilities and a target program. First, we focus on comparing the accuracy of VUDDY against SourcererCC, Deckard, and CCFinderX, which are not aimed at detecting "vulnerable" clones, and thus are not accurate when finding security vulnerabilities. After that, we will compare VUDDY with ReDeBug, which is designed for detecting vulnerable code clones.

To evaluate accuracy on the most equitable basis possible, we decided to conduct clone detection using each technique, then manually inspect every reported clone. The result of clone detection between our vulnerability database and Apache HTTPD 2.4.23 (352 KLoC) is shown in Table 3. As it is very challenging to find literally every vulnerability (including unknown vulnerabilities) in the target program, we cannot easily determine false negatives of tested techniques. To be clear, values of the FN column in Table 3 only account for

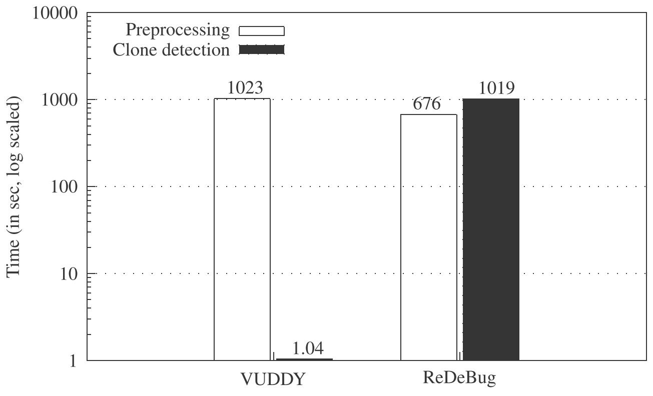


Fig. 6 – Preprocessing and clone detection time of VUDDY and ReDeBug, targeting Android firmware (14.86 MLoC, 349 K functions).

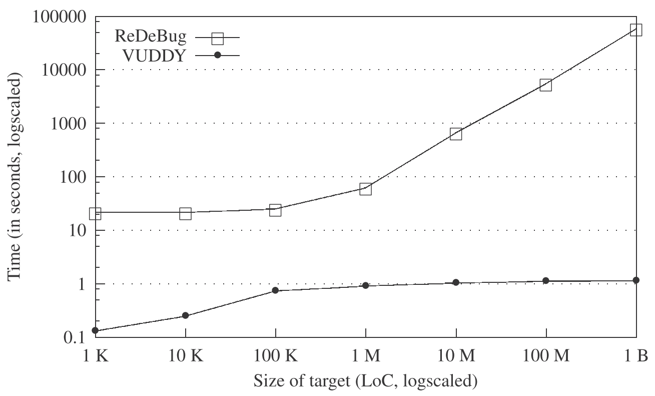


Fig. 7 – Clone detection time of VUDDY and ReDeBug, when varying size of target programs are given.

Table 3 – Accuracy of VUDDY, SourcererCC, Deckard, and CCFinderX when detecting clones between the vulnerability database and Apache HTTPD 2.4.23. Time is measured in seconds.

Technique	Time	Rep[†]	TP	FP	FN	Prec	Rec
VUDDY	22	9	9	0	2	1.00	0.82
SourcererCC (1.0)*	122	1	1	0	8	1.00	0.11
CCFinderX	1201	74	11	63	1	0.15	0.92
Deckard (1.0)*	58	57	3	54	8	0.05	0.27
SourcererCC (0.7)*	125	56	2	54	7	0.04	0.22
Deckard (0.85)*	234	462	4	458	8	0.01	0.33

* The values between parentheses denote minimum similarity threshold configuration.
† Denotes the number of clones each technique reported.

indisputable false negatives. For example, FN of VUDDY is the number of code clones detected by the other techniques that are not false positives, but not detected by VUDDY.

VUDDY reported 9 code clones in 22 seconds, and all of the findings were unpatched vulnerable clones in Apache HTTPD 2.4.23. This result indicates that function-level granularity and code abstraction have a positive effect on the accuracy of VUDDY. However, SourcererCC with 100% similarity setting also had precision of 1.0, but reported only one true positive case. It missed 8 vulnerable clones which VUDDY detected, because of its filtering heuristics. We lowered the minimum similarity threshold to 70%, expecting that SourcererCC might detect more true positive cases. However, it ended up detecting only two legitimate vulnerable clones, whereas introducing 54 false positives. The granularity unit of SourcererCC (i.e., bag-of-tokens) is too fine to be utilized for vulnerable code clone detection. Deckard with minimum similarity set to 100% reported 57 clones, and 54 cases were confirmed to be false positives. This shows that two perfectly matching abstract syntax trees (ASTs) are not necessarily generated from the same code fragments. Furthermore, when the minimum similarity threshold was set to 85%, Deckard detected only 4 true positive clones, with 458 false positives. This result accords with the observation of Jiang et al. (2007a) which claims that Deckard has 90% false positive. CCFinderX was the only technique that reported more true positive cases than those of VUDDY. This is because it takes advantage of a token-level granularity, which is the finest unit. However, 63 out of 74 reported clones were false positives, and CCFinderX required the most time to complete.

We analyzed the false positive cases of each tool, and discovered a fatal flaw of the compared techniques. In most of the false positive cases, SourcererCC, Deckard, and CCFinderX falsely identified patched functions in the target as clones of unpatched functions in the vulnerability database. We present one case in which patched benign function is identified as a clone of old, vulnerable version of the function, by all techniques but VUDDY. In Listing 3, we can observe that the statements removed and added by the patch are very similar. Eventually, the unpatched function and patched function have so similar structure and tokens that SourcererCC, Deckard, and CCFinderX misleadingly report them as a clone pair.

Listing 3: Snippet of the patch for CVE-2015-3183 which is already applied in request.c of Apache HTTPD 2.4.23.

```
- if (access_status == OK) {
- ap_log_rerror(APLOG_MARK, APLOG_TRACE3, 0, r,
- "request authorized without authentication by "
- "access_checker_ex hook: %s", r->uri);
- }
- else if (access_status != DECLINED) {
- return decl_die(access_status, "check access", r);
...
+ else if (access_status == OK) {
+ ap_log_rerror(APLOG_MARK, APLOG_TRACE3, 0, r,
+ "request authorized without authentication by "
+ "access_checker_ex hook: %s", r->uri);
+ }
+ else {
+ return decl_die(access_status, "check access", r);
```

We also analyzed the false negative cases. VUDDY did not detect two vulnerable functions that both SourcererCC (70% similarity threshold) and Deckard detected. The sole reason is that some lines of code, other than the vulnerable spot addressed by security patches, were modified in the function. We currently have vulnerable functions of the repository snapshots right before the security patches are applied. However, this is a trivial issue that can be easily resolved, because we can retrieve every different versions of a vulnerable function and add them in our database. For example, a command "`git log -p filename`" retrieves the entire change history of the queried file. Older snapshots of vulnerable functions are naturally obtained from the change history, and we can insert these into our vulnerability database. From a different standpoint, it is very surprising that SourcererCC and Deckard have more false negatives than VUDDY has. For these cases they failed to identify two identical functions as clones, implying that these techniques are not complete.

In summary, although Apache HTTPD is a moderately-sized project consisting of 350 KLoC, a lot of false positive cases are reported by techniques other than VUDDY. It is only logical that the bigger a target program is, the more false alarms are generated. Therefore, we confidently conclude that SourcererCC, Deckard, and CCFinderX are not suitable for detecting vulnerable clones from large code bases, as they will report so many false positive cases which cannot be handled by restricted manpower. Moreover, SourcererCC and Deckard had more false negatives than VUDDY had.

In contrast to SourcererCC, Deckard, and CCFinderX that solely aim for detecting any code clones and thus are not accurate when finding security vulnerabilities, ReDeBug is designed for detecting vulnerable code clones. Thus, we again conducted an in-depth comparison of VUDDY and ReDeBug in terms of accuracy.

False positive: VUDDY overwhelms ReDeBug with decisive margin, with respect to accuracy. As shown in Table 4, no false positive was reported by VUDDY. However, we conducted a manual inspection for 12 hours with 2090 code clones reported by ReDeBug to find that 1845 (88.3%) of these code clones were duplicates, because ReDeBug counts the number of CVE patches rather than the number of unpatched spots in the target code. After removing duplication, the number of clones reduced to 245. Then, we were able to find 43 (17.6%) false positives among the 245 unique code clones through a further inspection. The false positive cases were attributed to two causes: ReDeBug is language agnostic, and there is a technical limitation in their approach.

The language agnostic nature of ReDeBug causes the technique to find code clones of trivial patches (i.e., hardly related

Table 4 – Accuracy comparison of VUDDY and ReDeBug, targeting Android firmware (14.86 MLoC, 349 K functions).

	VUDDY	ReDeBug
# initial reports	206	2090
# multiple counts	0	1845
# unique clones	206	245
# false positives	0	43
# end result	206	202
# unique findings	25	21
# common findings	181	

to vulnerability), such as patches that modify macro statements, structs, and header inclusion or exclusion. For example, the patch for CVE-2013-0231 adds header inclusion statements to the beginning of `pciback_ops.c` in the xen driver of Linux kernel. The patch for CVE-2015-5257 adds an initialization statement of a struct member variable. Although ReDeBug found and reported that these patches are not applied in the Android smartphone, these unpatched codes cannot be vulnerabilities. On the other hand our mechanism targets only the functions, and therefore refrains from reporting such trivial code clones.

ReDeBug also has a technical limitation that contributes to the false positives. When ReDeBug processes the patches, it excludes the lines prefixed by a "+" symbol to obtain the original buggy code snippet, and then removes curly braces, redundant whitespaces and comments from the snippet. When searching for the snippet in the target source code, the lack of context leads to false positives. For example, ReDeBug reported a benign function in `xenbus.c` as an unpatched vulnerability, where the patch actually adds a line of comment to the original source code without making any significant changes to other lines of code. Even worse, ReDeBug erroneously detected the `nr_recvmsg` function shown in Listing 4, although the corresponding patch in Listing 5 is already applied. In this case, the sequence of lines 3, 6, 8, and 9 in the patch exactly matches lines 3, 4, 6, and 7 of the function in Listing 4 after preprocessing. This example reveals the limitation of a line-level granularity, responsible for causing false positives.

Listing 4: nr_recvmsg function in Android firmware which is erroneously reported as vulnerable by ReDeBug.

```
1 sax->sax25_family = AF_NETROM;
2 skb_copy_from_linear_data_offset(skb, 7, sax->sax25_call.
      ax25_call,
3 AX25_ADDR_LEN);
4 msg->msg_namelen = sizeof(*sax);
5 }
6 skb_free_datagram(sk, skb);
7 release_sock(sk);
```

Listing 5: Patch for CVE-2013-7266.

```
1 sax->sax25_family = AF_NETROM;
2 skb_copy_from_linear_data_offset(skb, 7, sax->sax25_call.
      ax25_call,
3 AX25_ADDR_LEN);
4 + msg->msg_namelen = sizeof(*sax);
5 }
6 - msg->msg_namelen = sizeof(*sax);
7 -
8 skb_free_datagram(sk, skb);
9 release_sock(sk);
```

False negative: Table 4 shows the number of unique findings of VUDDY and ReDeBug, which represent the false negatives of each other. In terms of false negatives, VUDDY and ReDeBug are complementary. Owing to the abstraction, VUDDY was able to find 25 vulnerable code clones in which data types, parameters, variable names, and function's names were modified. However, ReDeBug was not resilient to such changes. One of the cases is the function in Listing 7, which should have been patched by Listing 6 but not. While the security patch is not applied, a `const` qualifier is inserted in line 1 of Listing 7. ReDeBug tries to detect the window consisting of lines 1 to 6, and fails because of `const`. However, VUDDY is capable of detecting such variant of vulnerable function because both `const wlc_ssid_` and `wlc_ssid_t` are replaced with DTYPE after applying abstraction.

Listing 6: Patch for CVE-2016-2493.

```
ssid = (wlc_ssid_t *) data;
memset(profile->ssid.SSID, 0,
sizeof(profile->ssid.SSID));
+ profile->ssid.SSID_len = MIN(ssid->SSID_len,
      DOT11_MAX_SSID_LEN);
memcpy(profile->ssid.SSID, ssid->SSID, ssid->SSID_len);
profile->ssid.SSID_len = ssid->SSID_len;
break;
```

Listing 7: Vulnerable function in kernel/drivers/net/wireless/bcmdhd4359/wl_cfg80211.c

```
1 ssid = (const wlc_ssid_t *) data;
2 memset(profile->ssid.SSID, 0,
3 sizeof(profile->ssid.SSID));
4 memcpy(profile->ssid.SSID, ssid->SSID, ssid->SSID_len);
5 profile->ssid.SSID_len = ssid->SSID_len;
6 break;
```

The 21 cases VUDDY missed but ReDeBug detected resulted from the aforementioned reason: ReDeBug detected unpatched functions even if lines other than security patch addresses were modified, because it utilizes a line-level granularity. However, we emphasize again that these cases can be detected by VUDDY if we reinforce our vulnerability database by adding older snapshots of vulnerable functions.

After examining the wide discrepancies in speed and accuracy between VUDDY and ReDeBug, we concluded that VUDDY delivers results that are much more precise and accomplishes this with faster speed.

3.4. *Exact-matching vs abstract matching*

Our abstraction scheme enables VUDDY to detect variants of known vulnerabilities. We tested VUDDY with an Android firmware (14.9 MLoC). VUDDY reported 166 vulnerable clones without abstraction and 206 clones with abstraction. This means that VUDDY detects 24% more clones with abstraction, which are unknown vulnerabilities. We manually inspected the clones, and identified no false positive.

4. Empirical evaluation: Case study

In practice, a lot of software systems clone code from other software, and are exposed to vulnerabilities. Taking advantage of the scalability and accuracy of VUDDY, we were able to investigate a wide range of software projects. In this section, we empirically evaluate the utility of VUDDY by introducing various real-world software systems detected by VUDDY which are affected by cloned old vulnerabilities. Moreover, we analyze the root cause and draw an important insight that cloned vulnerabilities require a considerable amount of time to be patched, and this time lag is expanding the possible attack surface of various software systems.

According to the scale and cause of clones, we classify clone-induced vulnerabilities into the following three categories:

- Kernel clone cases,
- Library clone cases, and
- Intra-project clone cases.

4.1. *Kernel clone cases*

Linux kernel is one of the oldest and biggest open source projects. Its widespread use spans to a number of OS distributions such as Ubuntu, Debian, and recently, a lot of IoT and mobile devices adopt Linux kernel as a basis of their operating systems.

An important characteristic of the cases in which the kernel is reused, is that the reused kernel usually lags behind the latest kernel. This is very prevalent in the ecology of IoT devices including Android smartphones, Tizen appliances, and Linux-oriented operating systems. We observed that it often requires at least half a year to develop an operating system on the basis of a certain version of Linux kernel, which eventually leaves the end products (e.g., the IoT devices, OS distributions, and smart appliances running the Tizen OS) subject to the vulnerabilities which are reported during the period of development. In other words, kernel-based devices inevitably lag behind the patching efforts of Linux kernel developers, or a number of open source project participants.

We focused our analysis on Android smartphones which take a majority of market share in 2016 and 2017: Smartphone A and Smartphone B.[1] VUDDY detected several kernel vulnerabilities from these smartphones, and some are patched after we reported these vulnerabilities to the manufacturers. Our major findings of kernel clone cases are as follows:

- CVE-2016-5195 from Smartphone A: Race condition allowed read and write of files without permission. Actual device exploited.
- CVE-2017-7472 from Smartphone A and B: Memory leak vulnerability caused kernel panic. Actual device exploited.
- CVE-2017-8266 from Smartphone B: Use-after-free bug caused kernel panic. Actual device exploited.

CVE-2016-5195 (Dirty COW) in Smartphone A. Dirty COW vulnerability was once discovered and fixed by a Linux kernel developer in 2005, but its fix was reverted, thereby nullifying the initial fix. VUDDY detected the vulnerable clone of Dirty COW in the unpatched, up-to-date firmware (at the time) of Smartphone A, and we successfully exploited the vulnerable clone to gain root privilege of the smartphone. If VUDDY had been employed to find known old vulnerabilities before the affected kernels were released, Linux could have prevented such brutal vulnerability from being propagated through a number of OS distributions including the one shipped with Smartphone A which hold more than half of the market share at the time.

[1] Names of the smartphones are anonymized because of legal issues.

CVE-2017-7472 in Smartphone A and B. CVE-2017-7472 is a memory leak vulnerability which affects Linux kernels before 4.0.13. Although the patch was released in April 2017, VUDDY found that the vulnerability still resides in the latest firmware of both smartphones. The aforementioned problem of time difference between kernel release and smartphone release can be found in this case, as well: the kernel versions in these smartphones are 3.18.20 (released in August 2015) and 3.18.14 (released in May 2015), respectively. To make things worse, the exploit for this vulnerability is open to the public in Metasploit DB, and it only has two lines of code in its main function (see Listing 8). By running this simple PoC code, we readily triggered a kernel crash in both phones and verified that this vulnerability is able to affect the real up-to-date devices.

Listing 8: PoC code for CVE-2017-7472.

```
#include <sys/types.h>
#include <keyutils.h>

int main() {
    for (;;)
        keyctl_set_reqkey_keyring(
            KEY_REQKEY_DEFL_THREAD_KEYRING);
}
```

CVE-2017-8266 in Smartphone B. Another vulnerability found in Smartphone B is a use-after-free bug triggered by a race condition. This case is slightly different from the previous case, because the origin of the vulnerability is not the original Linux kernel, but the Qualcomm's Android for MSM project.

Patch for this vulnerability adds two lines which locks and unlocks mutex respectively, before and after freeing a memory block. Although half a year has passed since this simple patch was applied to the MSM kernel by Codeaurora, Smartphone B is still vulnerable. The PoC is available at GitHub repository of the person who reported the vulnerability, and we were able to make Smartphone B's kernel crash with the PoC that triggers a race condition.

Fig. 8 illustrates a typical multiple-step code cloning case, of Smartphone A. We can observe that the kernel is 1 year and 8 months behind the release of the firmware. Device manufacturers have to be aware of this temporal discrepancy, and try to

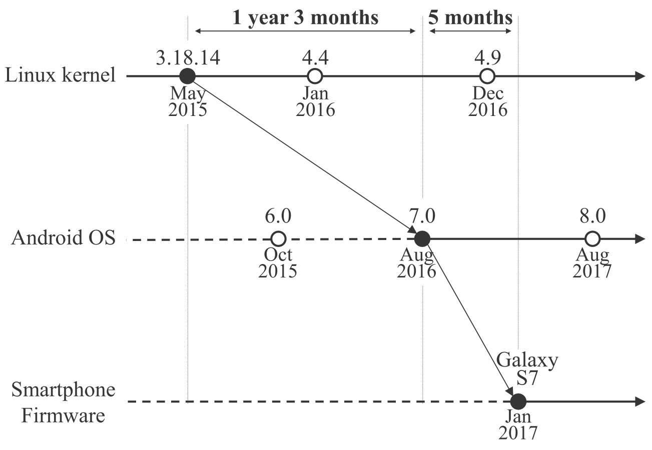


Fig. 8 – Multiple-step code cloning: Linux kernel, Android OS, and smartphone firmware.

narrow the gap as much as possible. Prompt update is a must, and in some cases where update is not a viable solution (e.g., we cannot replace an old kernel with a latest one easily), developers have to set up a channel for emergency patch release and make sure every n-day vulnerability is patched.

4.2. *Library clone cases*

In practice, library reuse takes place very frequently, as libraries are meant to be reused. Any software, small or large, can use libraries without much restriction. For example, the VLC media player is an open source media player in which at least 90 third-party libraries (including very popular ones such as FFmpeg, FLAC, LAME, libmpeg2, and QT5) are used. Consequently, many projects are prone to a wide range of vulnerabilities attributable to the outdated libraries they use.

We can classify library reuse patterns into three: 1) Full-source inclusion, 2) Dependency installation, and 3) Partial inclusion. Our major findings of library clone cases include:

- CVE-2016-3191 of PCRE library detetcted in MongoDB: Patched MongoDB was released 9 months after the patch had been released by the vendor of PCRE library.
- CVE-2012-0876 of Expat library detected in Apache HTTPD: Sending a crafted packet to server caused DoS and made system unavailable.
- CVE-2016-5161 and CVE-2016-5172 in SBrowser: Mobile web browser using an old chromium engine had serious vulnerabilities that could cause privacy issues.
- CVE-2011-3048 of LibPNG detected in SBrowser: Browser cloned LibPNG old, vulnerable version which was 3 years and 9 months behind the latest version.

4.2.1. *Database engines*

PCRE in MongoDB. CVE-2016-3191 is a pattern-mishandling vulnerability in PCRE (Perl Compatible Regular Expressions) which allows remote attackers to cause buffer overflow and execute arbitrary code through crafted regular expressions. MongoDB is the most widely-used NoSQL database engine.

We can observe an important vulnerability patching procedure in PCRE and MongoDB. Initially, the patch for CVE-2016-3191 was released on February 10, 2016. The patched version, PCRE 8.39, was officially released on June 14, 2016. And then, the updated PCRE was integrated to MongoDB 3.0.13 on October 31, 2016, and MongoDB 3.2.11 on November 18, 2016. In other words, it required approximately 9 months for a brutal vulnerability to be patched in one of the most popular database programs, even though its cause, effect, PoC (Proof of Concept) and the patch are disclosed to the public.

In this sense, attackers do not have to put a lot of effort into zero-day vulnerability detection. They have more than half a year to try exploiting popular software projects with already disclosed zero-day (i.e., n-day) vulnerabilities.

Fig. 9 illustrates how much time is required for an update in PCRE to be applied in MongoDB. The primitive versions of MongoDB cloned PCRE 7.4, which was released in September 2007. Although PCRE had been updated several times (there had been 15 official PCRE releases), the changes were not applied to MongoDB until version 2.1.0 which was released in February 2012. After four years of delayed update, MongoDB

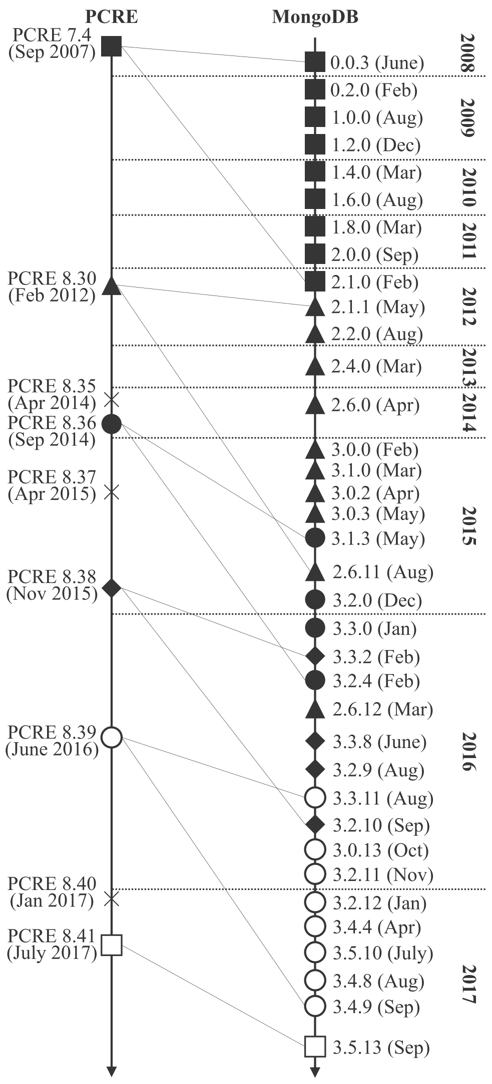


Fig. 9 – Temporal relationship between various versions of PCRE and MongoDB. Each dotted horizontal line makes a division between years. Markers of the same shape represent which version of PCRE is cloned into which version of MongoDB. For example, MongoDB 3.1.3, 3.2.0, 3.3.0, and 3.2.4 cloned PCRE 8.36 (denoted by filled circle markers).

2.1.1 was finally shipped with an updated PCRE (version 8.30, released in February 2012). This delay is repeated in the later versions of MongoDB. Even though PCRE 8.36 was released in September 2014, MongoDB kept using PCRE 8.30 until May 2015. Moreover, although the update was applied in MongoDB version 3.1.3 in May 2015, MongoDB 2.6.11 was released in August 2015 with old PCRE 8.30 rather than 8.36. This shows that branches of software can have a bad influence on prompt updates of the libraries used, because of consistency and dependency issues. Afterwards, PCRE 8.38 had a three-month delay (from November 2015 to February 2016) to be updated in MongoDB, and PCRE 8.39 had a two-month delay. PCRE 8.40 was not even considered to be updated, and two months after the latest PCRE 8.41 was released, MongoDB updated its library in 3.5.13.

4.2.2. *Web servers and browsers*

Expat library in Apache HTTP Server. In the Apache HTTP server, VUDDY discovered a vulnerable code clone of CVE-2012-0876, which eventually turned out to be a zero-day vulnerability. (Zero-day herein follows the typically accepted definition: *a vulnerability that is unknown to those who would be interested in mitigating the vulnerability*. Apache was not aware that this vulnerability existed in their HTTP daemon, and we could exploit this vulnerability to compromise the system.) The versions before 2.4.25 (released in December 2016) are affected. Apache HTTP server includes Apache Portable Runtime (APR) project which provides several APIs. One of the features Apache HTTP Server makes use of is the XML parser of the Expat library included in the APR project. Unfortunately, the version in APR is outdated and is vulnerable to CVE-2012-0876, a so-called Hash DoS attack.

We could use a specially crafted XML file to trigger the vulnerability, and force the Apache HTTP server daemon to consume 100% of CPU resources. Listing 9 shows part of the patches for CVE-2012-0876, and Listing 10 is an excerpt of the vulnerable function in Apache HTTP server, which can be triggered with a crafted packet to cause DoS.

Listing 9: Patch for CVE-2012-0876 retrieved from Google Android repository.

```
 for (i = 0; i < table->size; i++)
   if (table->v[i]) {
- unsigned long newHash = hash(table->v[i]->name);
+ unsigned long newHash = hash(parser, table->v[i]->name);
    size_t j = newHash & newMask;
```

Listing 10: A five-year-old vulnearbility (CVE-2012-0876) in Apache HTTPD 2.4.23 (released in 2017).

```
...
for (i = 0; i < table->size; i++)
  if (table->v[i]) {
    unsigned long newHash = hash(table->v[i]->name);
    size_t j = newHash & newMask;
    step = 0;
...
```

We reported this zero-day vulnerability, which could critically affect numerous web services that run Apache HTTP server, and the Apache security team confirmed this vulnerability. To resolve the problem, they took two measures. Firstly, they excluded the source code of APR from the Apache HTTP Server's repository. In other words, they switched the library reuse pattern from "Full-source inclusion" to "Dependency installation", rather than integrating the upstream APR to their source code. This would be expected to accelerate the process of library updates. Secondly, they excluded Expat sources from APR 1.6.

WebKit and V8 vulnerabilities in SBrowser. Smartphones manufactured by S-electronics are shipped with a built-in web browser, named SBrowser. Like many other web browsers, SBrowser is based on Google Chromium, an open source browser project. Naturally, vulnerabilities in Chromium project are propagated to SBrowser, and not being patched even in the latest version (SBrowser 5.4.21.54, September 2017).

One of the vulnerabilities is CVE-2016-5161, a type confusion vulnerability in WebKit. To understand how this 1.5-year-old vulnerability sneaked into the latest mobile browser, an understanding of the genealogy of SBrowser is required. WebKit is Apple's web browser engine used by a lot of browsers including old Chrome. In 2013, Google forked WebKit to make their own rendering engine, which is Blink. Chromium is powered by Blink, and SBrowser inherited this characteristic. We successfully exploited CVE-2016-5161 in the SBrowser and caused a denial of service.

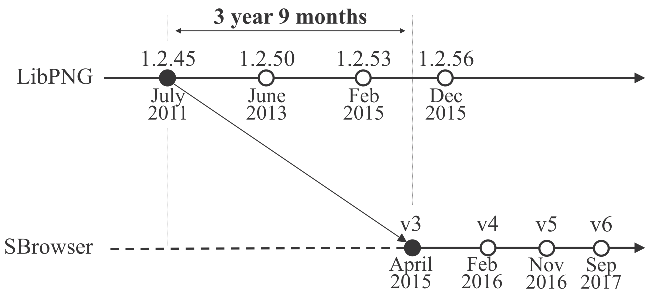


Fig. 10 – Single-step code cloning: LibPNG and SBrowser.

Another vulnerability is CVE-2016-5172, a scope-related vulnerability in Google V8. VUDDY detected that this vulnerability is not patched yet in SBrowser. By the use of PoC which is already disclosed, we could make SBrowser crash, and hijack its session information.

A common characteristic of both of the vulnerabilities is that, the PoC can be easily found on the web. In other words, SBrowser has been exposed to vulnerabilities of which PoCs are accessible through web for more than a year and a half.

LibPNG in SBrowser. In September 2016, VUDDY detected another old vulnerability, CVE-2011-3048, in the older version (up-to-date at the time) of the SBrowser. The vulnerability stemmed from the use of an outdated LibPNG library: version 1.2.45, which was released in July 2011. VUDDY detected that the fix for CVE-2011-3048 is not applied in that version, leaving the browser vulnerable. We consider this case to be very alarming because a vulnerability which had already been patched five years ago was still being distributed through widely-used smartphones. After we reported this bug to the manufacturers, they affirmed that they would conduct a dependency check and library update for the next release. Finally, this vulnerability was patched and the up-to-date browser is using LibPNG 1.2.56, which patched the vulnerability.

Fig. 10 depicts the single-step code cloning case of SBrowser. A lot of instances of library reuse fall into the category of single-step code cloning, where outdated versions of libraries are included in the project. Software developers have to carefully keep track of the libraries used in their projects to avoid exposing their software to n-day vulnerabilities and corresponding attacks.

4.2.3. *Others*

PCRE in Extempore. Extempore is a cyber-physical programming environment. Its repository in GitHub has received 922 stars, and the project is very actively managed. However, the version of the PCRE library it uses is still 8.38, which is vulnerable to CVE-2016-3191.

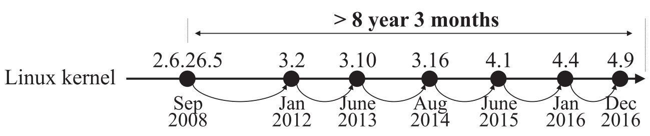


Fig. 11 – Intra-project code cloning: Recurring Linux kernel vulnerability.

PCRE in WinMerge. WinMerge is a famous Windows tool for visual difference display and merging. Its latest release was downloaded more than 33,000 times this week, and more than 1.1 million times since January 2017 to September 2017. In contrast to its reputation, this program is also vulnerable to CVE-2016-3191 because of the use of the outdated PCRE library.

LibPNG in the Visualization Toolkit (VTK). VTK is a system for 3D computer graphics, image processing, and visualization. VTK's GitHub repository has received 696 stars, and is forked 457 times. However, VUDDY detected CVE-2015-8540 (CVSS 9.3) in the LibPNG 1.0.65 in VTK repository. By exploiting this underflow read bug via a crafted PNG image, we caused an out-of-bounds read.

4.3. *Intra-project code clone cases*

Owing to our abstraction strategy, we detected an 8-year-old vulnerability (CVE-2008-3528) which possibly is a zero-day vulnerability, in the latest stable Linux kernel. Very interestingly, although the original vulnerability was found in ext2, ext3, and ext4 file systems of the kernel 2.6.26.5, and then patched in 2008, the nilfs2 file system of which the implementation is very similar (but differs in relation to some identifiers) to that of ext2 has remain unpatched to date (see Fig. 11). The problematic function *`nilfs_dotdot` is in `linux/fs/nilfs2/dir.c`.

Listing 11: Original patch of CVE-2008-3528 targeting ext2 file system of Linux.

```
struct ext2_dir_entry_2 * ext2_dotdot (struct inode *dir,
       struct page **p)
{
- struct page *page = ext2_get_page(dir, 0);
+ struct page *page = ext2_get_page(dir, 0, 0);
   ext2_dirent *de = NULL;

   if (!IS_ERR(page)) {
```

Listing 12: Buggy function in nilfs2 file system of Linux.

```
struct nilfs_dir_entry *nilfs_dotdot(struct inode *dir,
       struct page **p)
{
   struct page *page = nilfs_get_page(dir, 0);
   struct nilfs_dir_entry *de = NULL;

   if (!IS_ERR(page)) { de = nilfs_next_entry( ...
```

The function described in Listing 12 is suspected to be cloned from the implementation of the ext2 file system, because file systems share a considerable amount of similar characteristics. Even though the name of the function called at line 3 of Listing 12 is different from that of the original buggy function (`ext2_get_page`) in Listing 11, this is detected by VUDDY because as described in section 2, we abstract the function calls by replacing the names of the called function with FUNCCALL.

The contents of function `ext2_get_page` and `nilfs_get_page` are also identical except for their names and a few identifiers, and thus we attempted to trigger the vulnerability in Ubuntu 16.04 which is built upon kernel version 4.4. Surprisingly, we could trigger the "printk floods" vulnerability which in turn causes denial of service, by mounting a corrupted image of the nilfs2 file system. We contacted a security officer of Redhat Linux, and he confirmed that this vulnerability should be patched. This case shows that VUDDY is capable of detecting unknown variants of known vulnerability.

4.4. *Lessons learned*

In this section we summarize and discuss the lessons learned from the case studies. As seen in the cases of the real-world vulnerable code clones, attack vector lies in a majority of programs that clone code from other software.

- A considerable number of old vulnerabilities tend to remain unpatched in the software systems that contain any kind of code clones (e.g., kernel clone, library clone, or intra-project clone). In general, these vulnerabilities live for 0.5–1.5 year, but some software have older vulnerabilities which are more than 3 years old.
- The cloned old vulnerabilities can be easily triggered by known exploits, and can cause severe damage.
- Patching the old vulnerabilities is challenging especially when multiple parties are involved in the clone chain. In addition, it is very hard to manually manage patches for every cloned component (e.g., software libraries).
- An automated analysis system is required to detect and handle these vulnerabilities. VUDDY is a system that maximized the scalability and accuracy.

5. Application: IoTcube

VUDDY has been serviced online for free (at IoTcube (Kim et al., 2017b), https://iotcube.net, Fig. 12) since April 2016, facilitating scalable and accurate inspection of software. Users of our service include commercial software developers, open source committers, and IoT device manufacturers. Please note that all the findings presented in the case study of section 4 are selected from the results gathered from the IoTcube system. `hmark` (Fig. 13) is the implementation of VUDDY, which pre-processes a target program and generates an index file. When a user uploads the generated index file to IoTcube, it shows the result.

The result page includes the number of detected vulnerable functions and CVEs (Fig. 14), the original repositories of the clones (Fig. 15), yearly distributions of CVEs (Fig. 16), CWE (Common Weakness Enumeration) distribution (Fig. 17), CVSS (Common Vulnerability Scoring System) score distribution (Fig. 18), and a tree view with which users are able to locate the files affected by vulnerable clones. Using the information provided in the tree view (Fig. 19) users are able to patch

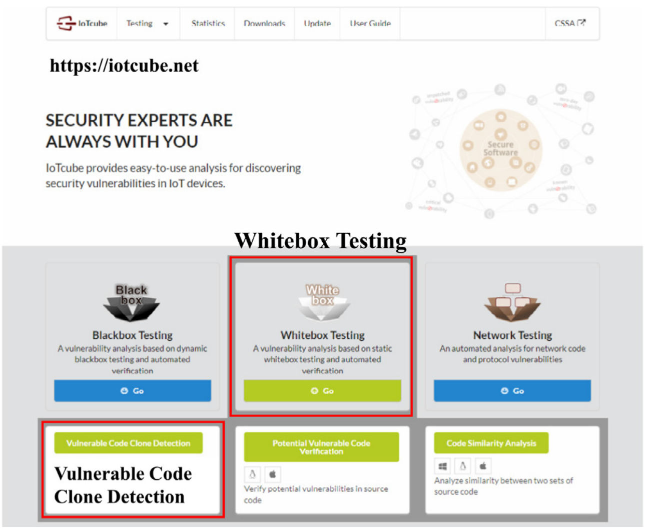
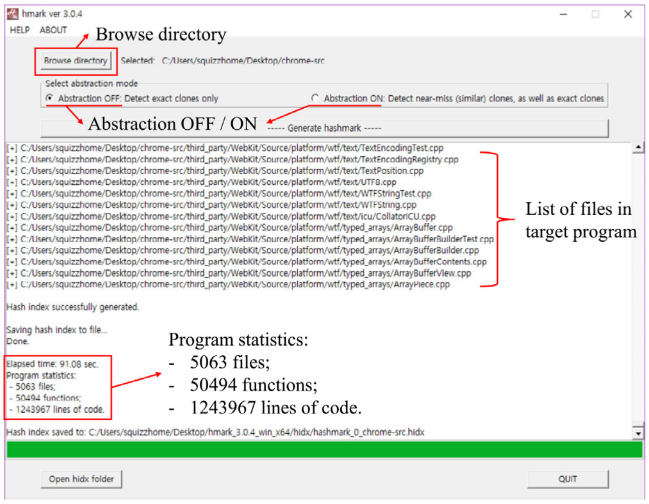

IoTcube
Testing
Statistics
Downloads
Update
User Guide
CSSA
https://iotcube.net
SECURITY EXPERTS ARE ALWAYS WITH YOU
IoTcube provides easy-to-use analysis for discovering security vulnerabilities in IoT devices.
Secure Software
Whitebox Testing
Black box
Blackbox Testing
A vulnerability analysis based on dynamic blackbox testing and automated verification
Go
White box
Whitebox Testing
A vulnerability analysis based on static whitebox testing and automated verification
Go
Network Testing
An automated analysis for network code and protocol vulnerabilities
Go
Vulnerable Code Clone Detection
Vulnerable Code Clone Detection
Potential Vulnerable Code Verification
Verify potential vulnerabilities in source code
Code Similarity Analysis
Analyze similarity between two sets of source code

Fig. 12 – The main page of IoTcube. Vulnerable code clone detection is under the Whitebox Testing menu.

the vulnerabilities, or report to the manufacturers for official actions.

Since the release of IoTcube in April 2016, 22 BLoC have been queried. VUDDY detected 279 K vulnerable code clones which include the cases described in section 4. CVE-2015-2695, a high-severity vulnerability in Kerberos5, was detected the most, and CWE-119 (Improper Restriction of Operations within the Bounds of a Memory Buffer), CWE-264 (Permissions, Privileges, and Access Controls), and CWE-377 (Insecure Temporary File) were the top three CWEs.

IoTcube is making a real impact on the world. One of the device manufacturers integrated IoTcube into its software development process to manage the vulnerabilities detected in its devices. Whenever a developer commits a code, the code is checked by VUDDY engine. If a vulnerability is detected, the code is automatically rejected by the system. Through such application, the manufacturer is preventing an accumulation of vulnerabilities.

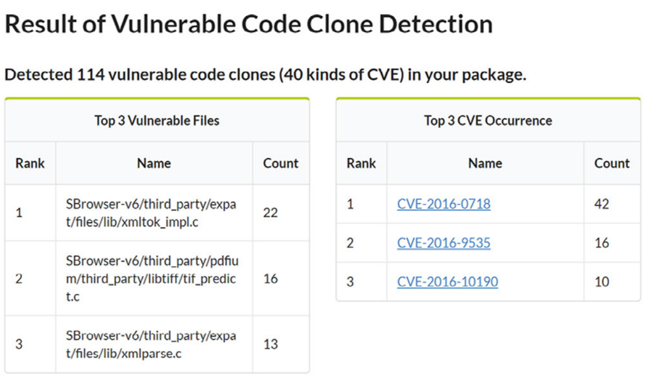


Selected: C:/Users/squizzhome/Desktop/chrome-src

Select abstraction mode
[✓ Abstraction OFF: Detect exact clones only] Abstraction ON: Detect near-miss (similar) clones, as well as exact clones

```
[+] C:/Users/squizzhome/Desktop/chrome-src/third_party/WebKit/Source/platform/wtf/text/TextEncodingTest.cpp
[+] C:/Users/squizzhome/Desktop/chrome-src/third_party/WebKit/Source/platform/wtf/text/TextEncodingRegistry.cpp
[+] C:/Users/squizzhome/Desktop/chrome-src/third_party/WebKit/Source/platform/wtf/text/TextPosition.cpp
[+] C:/Users/squizzhome/Desktop/chrome-src/third_party/WebKit/Source/platform/wtf/text/UTF8.cpp
[+] C:/Users/squizzhome/Desktop/chrome-src/third_party/WebKit/Source/platform/wtf/text/WTFStringTest.cpp
[+] C:/Users/squizzhome/Desktop/chrome-src/third_party/WebKit/Source/platform/wtf/text/WTFString.cpp
[+] C:/Users/squizzhome/Desktop/chrome-src/third_party/WebKit/Source/platform/wtf/text/icu/CollatorICU.cpp
[+] C:/Users/squizzhome/Desktop/chrome-src/third_party/WebKit/Source/platform/wtf/typed_arrays/ArrayBuffer.cpp
[+] C:/Users/squizzhome/Desktop/chrome-src/third_party/WebKit/Source/platform/wtf/typed_arrays/ArrayBufferBuilderTest.cpp
[+] C:/Users/squizzhome/Desktop/chrome-src/third_party/WebKit/Source/platform/wtf/typed_arrays/ArrayBufferBuilder.cpp
[+] C:/Users/squizzhome/Desktop/chrome-src/third_party/WebKit/Source/platform/wtf/typed_arrays/ArrayBufferContents.cpp
[+] C:/Users/squizzhome/Desktop/chrome-src/third_party/WebKit/Source/platform/wtf/typed_arrays/ArrayBufferView.cpp
[+] C:/Users/squizzhome/Desktop/chrome-src/third_party/WebKit/Source/platform/wtf/typed_arrays/ArrayPiece.cpp

Hash index successfully generated.

Saving hash index to file...
Done.

Elapsed time: 91.08 sec.
Program statistics:
 - 5063 files;
 - 50494 functions;
 - 1243967 lines of code.

Hash index saved to: C:/Users/squizzhome/Desktop/hmark_3.0.4_win_x64/hidx/hashmark_0_chrome-src.hidx
```

Fig. 13 – The preprocessor, called `hmark`. `hmark` preprocesses target program and generates an index file.



Result of Vulnerable Code Clone Detection

Detected 114 vulnerable code clones (40 kinds of CVE) in your package.

Top 3 Vulnerable Files

Rank	Name	Count
1	SBrowser-v6/third_party/expat/files/lib/xmltok_impl.c	22
2	SBrowser-v6/third_party/pdfium/third_party/libtiff/tif_predict.c	16
3	SBrowser-v6/third_party/expat/files/lib/xmlparse.c	13

Top 3 CVE Occurrence

Rank	Name	Count
1	CVE-2016-0718	42
2	CVE-2016-9535	16
3	CVE-2016-10190	10

Fig. 14 – Result page on IoTcube. When an index file of a smartphone firmware is uploaded, IoTcube detected 418 vulnerable code clones.

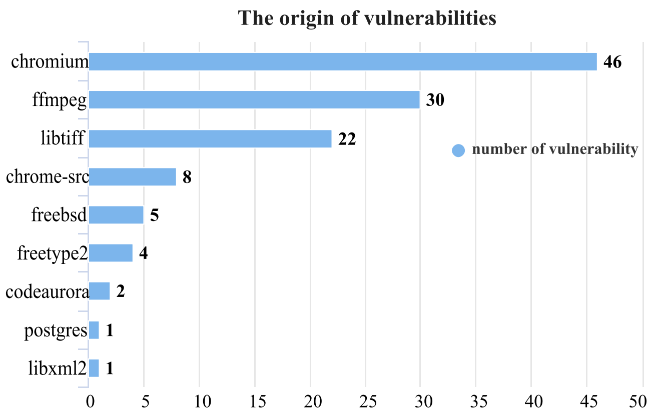


Fig. 15 – The origin of vulnerabilities. Vulnerabilities already reported and patched at Chromium, FFMpeg, LibTiff, and so on are detected in the mobile web browser application.

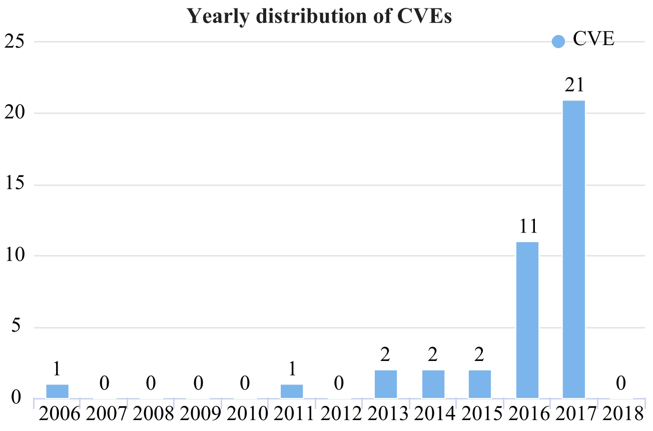


Fig. 16 – Yearly distribution of CVEs. As the tested browser application uses Chromium which was released in May 2015, many of the vulnerabilities exposed in 2016 and 2017 are not patched yet.

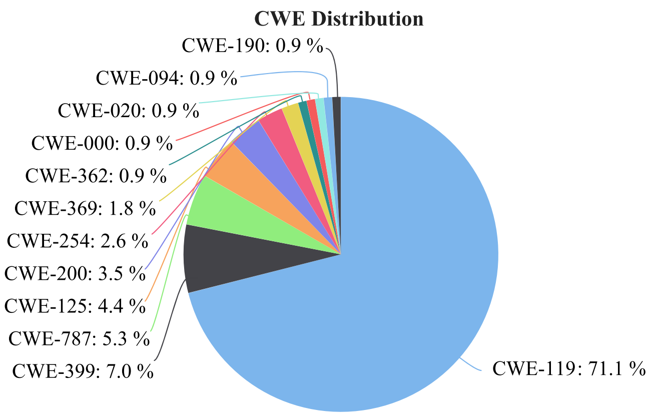


Fig. 17 – The distribution of CWEs. CWE-119 (memory buffer related vulnerability) is dominant in the browser application.

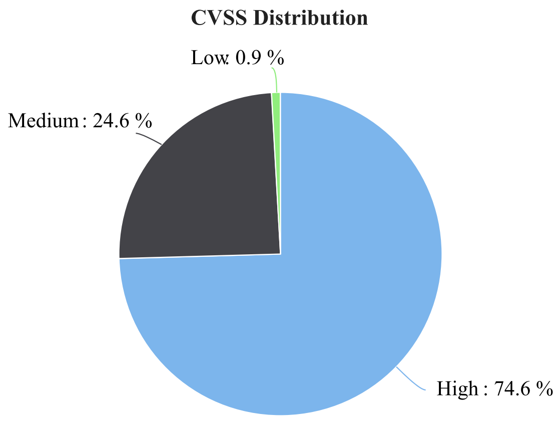


Fig. 18 – The distribution of CVSS. Over 74% of the detected CVEs are assigned with high severity score.

6. Discussion

6.1. *Detecting semantic clones*

Identifying semantic clones (i.e., the code fragments that are syntactically different but perform the same functionalities) is a complicated task that requires an in-depth scrutiny of a code or a binary object, and thus it is beyond the scope of this paper. For example, BLEX (Egele et al., 2014) measures the similarity score of two functions by dynamically instrumenting them under various environments (e.g., concrete values of the registers). As most dynamic approaches do, BLEX suffers from a low scalability, requiring 30 CPU minutes for measuring similarity of two versions of `ls` utility. Esh (David et al., 2016) performs static analysis to determine the similarity of binaries. Although scalability is not a big issue, this approach is not sensitive to small changes, and thus is not capable of clearly distinguishing an unpatched function from a patched function.

Rather, we specifically focus on making the approach scalable enough to handle massively growing software systems yet being accurate by relaxing the problem of code clone detection to only exact (Type-1) and renamed (Type-2) clones. As shown in section 3, VUDDY can provide non-trivial complementary benefits that existing approaches fail to provide: unrivaled speed, zero false positive rate, and low false negative rate, which eventually lead to good practical impacts.

6.2. *Collecting vulnerable functions*

In subsection 2.3, we addressed an automated way to collect vulnerable functions. This method has two limitations: the repository must be managed through Git or SVN, and the vulnerability patch must contain string “CVE-20” in the commit

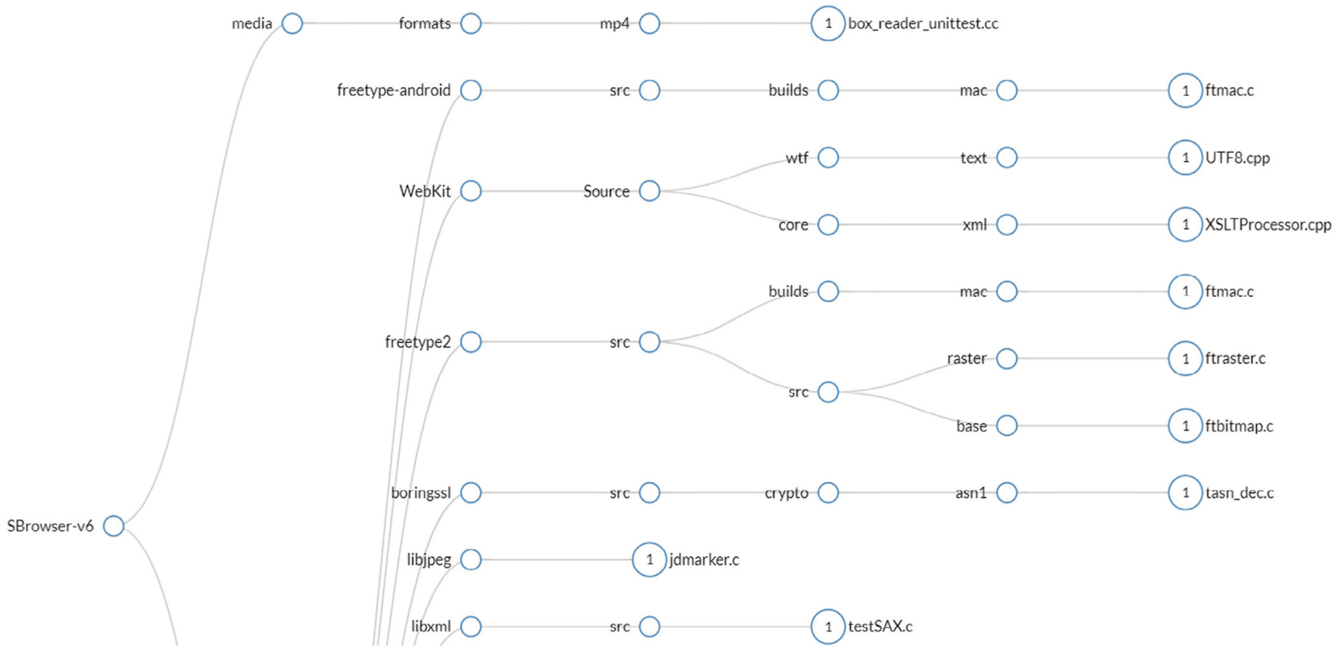


Fig. 19 – A tree view. Detailed information regarding the paths to the vulnerable functions and the corresponding security patches is provided.

log. However, numerous vulnerability patches are committed without a hint of CVE id. Therefore, we are trying to collect vulnerable functions from the channels other than git repositories, such as issue/bug trackers. As the format, or internal rules differ greatly depending on the issue trackers, automatizing the collection of vulnerabilities is suggested as a future work.

6.3. *Language support*

The current version of VUDDY supports C and C++. Since there are also considerable number of programs (and corresponding vulnerabilities) written in a variety of programming languages, a support for more languages is required. We leave this as a future work.

7. Conclusion

In this paper, we proposed VUDDY, which is an approach for scalable and accurate vulnerable code clone discovery. The design principles of VUDDY is directed toward extending scalability through function-level granularity and a length filter, while maintaining accuracy so that it can afford to detect vulnerable clones from the rapidly expanding pool of open source software. VUDDY adopts a vulnerability-preserving abstraction scheme which enables it to discover 24% more unknown variants of vulnerabilities. We implemented VUDDY to demonstrate its efficacy and effectiveness. The results show that VUDDY can actually detect numerous vulnerable clones from a large code base with unprecedented scalability and accuracy. In the case study, we presented several cases discovered by VUDDY, in which vulnerable functions remain unfixed for years and propagate to other programs.

Tremendous number of vulnerable code fragments will continue to be propagated to countless programs and devices. We strongly believe that VUDDY is a must-have approach to be used for securing various software when scalability and accuracy is required. To secure software or devices, software developers and device manufacturers have to put their best efforts to patch every vulnerable code clone reported by VUDDY. As another solution, we suggest that various version control systems or systems for project management (e.g., GitHub) establish a warning system by which a change or update in the original repository is notified to the projects that cloned the repository. This will immediately inform the developers of the vulnerabilities in the cloned libraries, and thus shorten the time between patch release and deploy.

Acknowledgment

This work was supported by Institute for Information & communications Technology Promotion(IITP) grant funded by the Korea government (MSIP) (No.R0190-16-2011, Development of Vulnerability Discovery Technologies for IoT Software Security, and No.2017-0-00184, Self-Learning Cyber Immune Technology Development).

REFERENCES

Baxter ID, Yahin A, Moura L, Anna MS, Bier L. Clone detection using abstract syntax trees. In: Proceedings of the International Conference on Software Maintenance. IEEE; 1998. p. 368–77.

Bellon S, Koschke R, Antoniol G, Krinke J, Merlo E. Comparison and evaluation of clone detection tools. IEEE Trans Softw Eng 2007;33(9):577–91.

Blackduck. 2016 the future of open source.

Dang Y, Zhang D, Ge S, Huang R, Chu C, Xie T. Transferring code-clone detection and analysis to practice. In: IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP). IEEE; 2017. p. 53–62.

David Y, Partush N, Yahav E. Statistical similarity of binaries. ACM SIGPLAN Not 2016;51(6):266–80.

Egele M, Woo M, Chapman P, Brumley D. Blanket execution: Dynamic similarity testing for program binaries and components. USENIX, 2014.

GitHub. About. Available from http://github.com/about. [Accessed 31 August 2017].

Godfrey MW, Tu Q. Evolution in open source software: a case study. In: Proceedings of the International Conference on Software Maintenance. IEEE; 2000. p. 131–42.

Jang J, Agrawal A, Brumley D. ReDeBug: finding unpatched code clones in entire os distributions. In: IEEE Symposium on Security and Privacy (SP), 2012. IEEE; 2012. p. 48–62.

Jiang L, Su Z, Chiu E. Context-based detection of clone-related bugs. In: Proceedings of the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on the foundations of software engineering. ACM; 2007a. p. 55–64.

Jiang L, Misherghi G, Su Z, Glondu S. DECKARD: Scalable and accurate tree-based detection of code clones. In: Proceedings of the 29th international conference on Software Engineering, IEEE Computer Society; 2007b. p. 96–105.

Kamiya T, Kusumoto S, Inoue K. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. IEEE Transactions on Software Engineering, 28(7); 2002. p. 654–70.

Kim S, Woo S, Lee H, Oh H. VUDDY: a scalable approach for vulnerable code clone discovery. In: 2017 IEEE Symposium on Security and Privacy (SP). IEEE; 2017a. p. 595–614.

Kim S, Woo S, Lee H, Oh H. Poster: IoTcube: an automated analysis platform for finding security vulnerabilities. 2017 IEEE Symposium on Poster presented at Security and Privacy (SP). IEEE, 2017b.

Koschke R. Survey of research on software clones. Dagstuhl Seminar Proceedings, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

Li H, Kwon H, Kwon J, Lee H. CLORIFI: software vulnerability discovery using code clone verification, concurrency and computation: practice and experience; 2015. p. 1900–17.

Li Z, Lu S, Myagmar S, Zhou Y. CP-Miner: finding copy-paste and related bugs in large-scale software code. IEEE Transactions on Software Engineering, 32(3); 2006. p. 176–92.

Li Z, Zou D, Xu S, Jin H, Qi H, Hu J. VulPecker: an automated vulnerability detection system based on code similarity analysis. In: Proceedings of the 32nd Annual Conference on Computer Security Applications. ACM; 2016. p. 201–13.

Nappa A, Johnson R, Bilge L, Caballero J, Dumitras T. The attack of the clones: a study of the impact of shared code on vulnerability patching. In: 2015 IEEE Symposium on Security and Privacy. IEEE; 2015. p. 692–708.

Pham NH, Nguyen TT, Nguyen HA, Nguyen TN. Detection of recurring software vulnerabilities. In: Proceedings of the IEEE/ACM international conference on Automated software engineering. ACM; 2010. p. 447–56.
Rattan D, Bhatia R, Singh M. Software clone detection: a systematic review. Inf Softw Technol 2013;55(7):1165–99.
Roy CK, Cordy JR, Koschke R. Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. Sci Comput Program 2009;74(7):470–95.
Sajnani H, Saini V, Svajlenko J, Roy CK, Lopes CV. SourcererCC: scaling code clone detection to big-code. In: Proceedings of the 38th International Conference on Software Engineering. ACM; 2016. p. 1157–68.
Scacchi W. Understanding open source software evolution, software evolution and feedback. Theory Pract 2006;9:181–205.
Succi G, Paulson J, Eberlein A. Preliminary results from an empirical study on the growth of open source and commercial software products. In: EDSER-3 workshop. Citeseer; 2001. p. 14–15.
Svajlenko J, Roy CK. Evaluating modern clone detection tools. In: 2014 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE; 2014. p. 321–30.
Svajlenko J, Roy CK. Evaluating clone detection tools with BigCloneBench. In: 2015 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE; 2015. p. 131–40.
Tian K, Yao D, Ryder BG, Tan G. Analysis of code heterogeneity for high-precision classification of repackaged malware. In: Security and Privacy Workshops (SPW), 2016 IEEE. IEEE; 2016. p. 262–71.
Tian K, Yao DD, Ryder BG, Tan G, Peng G. Detection of repackaged Android malware with code-heterogeneity features, 2017.
Wang T, Harman M, Jia Y, Krinke J. Searching for better configurations: a rigorous approach to clone evaluation. In: Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering. ACM; 2013. p. 455–65.
White M, Tufano M, Vendome C, Poshyvanyk D. Deep learning code fragments for code clone detection. In: Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering. ACM; 2016. p. 87–98.
Yamaguchi F, Lottmann M, Rieck K. Generalized vulnerability extrapolation using abstract syntax trees. In: Proceedings of the 28th Annual Computer Security Applications Conference. ACM; 2012. p. 359–68.
Zhang M, Duan Y, Yin H, Zhao Z. Semantics-aware Android malware classification using weighted contextual API dependency graphs. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security. ACM; 2014. p. 1105–16.

Seulbae Kim received the B.S. degree in Computer Science and Engineering from Korea University, Seoul Korea, in 2016. Currently, he is an M.S. candidate in Department of Computer and Radio Communications Engineering, Korea University. His research interests include software security and vulnerability analysis.

Heejo Lee is a Professor in the Department of Computer Science and Engineering, Korea University, Korea. Before joining Korea University, he was at AhnLab, Inc. as the CTO from 2001 to 2003. From 2000 to 2001, he was a Post-doctorate Researcher at CERIAS Purdue University, and was a visiting professor at CyLab/CMU in 2010. He received his B.S., M.S., and Ph.D. degree in Computer Science and Engineering from POSTECH, Korea. He is an Editor of the Journal of Communications and Networks, and the International Journal of Network Management. He received the ISC^2 ISLA award of community service star in 2016.