VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery

Seulbae Kim, Seunghoon Woo, Heejo Lee*, Hakjoo Oh
Department of Computer Science and Engineering
Korea University
Seoul, Korea
{seulbae, seunghoonwoo, heejo, hakjoo_oh}@korea.ac.kr

***Abstract*—The ecosystem of open source software (OSS) has been growing considerably in size. In addition, code clones - code fragments that are copied and pasted within or between software systems - are also 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 hour and 17 minutes, after which it requires a few seconds to identify code clones. In addition, we designed a security-aware 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. In this study, we describe its principles and evaluate its efficacy and effectiveness by comparing it with existing mechanisms and presenting the vulnerabilities it detected. VUDDY outperformed four state-of-the-art code clone detection techniques in terms of both scalability and accuracy, and proved its effectiveness by detecting zero-day vulnerabilities in widely used software systems, such as Apache HTTPD and Ubuntu OS Distribution.

I. Introduction

During the last few years, the number of open source software (OSS) programs has increased at a rapid pace. Research published in literature showed that open source software projects have linear to quadratic growth patterns [1], [2], [3]. In practice, the number of registered open source projects in SourceForge [4] increased from 136 K to 430 K between October 2009 and March 2014. GitHub [5] announced that its 10 millionth repository had been created in December 2013, with most of the repositories being software projects. Meanwhile, they currently have over 85 million projects, in March 2017.

The considerable increase in the number of OSS programs has naturally led to an increase in software vulnerabilities caused by code cloning, thereby posing dire threats to the security of software systems. Code cloning, the act of copying and pasting portions of other software, can be useful if it is properly exploited [6], [7]. However, in practice, code cloning is often regarded as a bad programming practice because it can raise maintenance costs [8], reduce quality [9], [10], produce potential legal conflicts, and even propagate software vulnerabilities [11], [12], [13]. In particular, as OSS programs are widely used as codebase in software development, (e.g., libraries), code cloning is becoming one of the major causes of software vulnerabilities. For example, the OpenSSL Heartbleed vulnerability (CVE-2014-0160) has affected several types of systems (including websites, web servers, operating system distributions, and software applications), because the affected system either used the whole OpenSSL library or cloned some part of the library for use in their systems.

Moreover, the lifecycle of vulnerabilities exacerbates such problems. Even if a vendor were to release a 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 [14]. For example, in April 2016, the "Dogspectus" ransomware was disclosed. This ransomware exploits a bug in the Linux kernel named "futex local privilege escalation vulnerability" (CVE-2014-3153) to deliver drive-by-download malware to the mobile devices that run an unpatched Android operating system (versions 4.0.3 to 4.4.4). Another example, the Dirty COW vulnerability (CVE-2016-5195), which exploits a race condition for privilege escalation, was found in the memory subsystem of the Linux kernel in October 2016. What makes this vulnerability outrageous is that this bug was already fixed in 2005, but the fix was undone due to another problem raised by the fix. As shown in the examples, old, vulnerable code fragments that are supposed to be eliminated, are ceaselessly re-emerging in various locations for a variety of reasons.

Many researchers have proposed code clone detection techniques to address clone-related problems. However, to our surprise, few techniques are suitable for accurately finding vulnerability in a scalable manner. For example, lexical tech-

*Heejo Lee is the corresponding author.

niques such as CCFinder [15] have the disadvantage of high complexity as it uses a suffix tree algorithm to measure the similarity between token sequences of programs. In addition, its parameter replacement strategy is so aggressive that it introduces a significant number of false positives. Similarly, approaches that transform code into abstract data structures (e.g., abstract syntax trees) have to apply expensive tree-matching operations or graph mining techniques for similarity estimation [16], [17]. Although such an approach would be capable of discovering code fragments with similar syntactic patterns, this does not guarantee accurate vulnerability detection because two code fragments with identical abstract syntax trees do not necessarily contain the same vulnerability. Notable exceptions are ReDeBug [18] and SourcererCC [19]. ReDeBug aims to achieve both accuracy and scalability by applying hash functions to lines of code and later detecting clones by comparing hash values. However, as we show in this paper, ReDeBug is still not satisfactory both in terms of accuracy and scalability when it comes to finding vulnerable code clones in massive code bases. For example, when testing an Android smartphone (15 MLoC), ReDeBug requires half an hour, and has 17.6 % false positives. SourcererCC uses a bag-of-tokens strategy to manage minor to specific changes in clones, which impairs the accuracy from a security perspective. For example, SourcererCC detects clones in which the sequence of code is changed, or statements are inserted. As a result, it misleadingly detects a patched code fragment as a clone of an unpatched code fragment.

In this paper, we present VUDDY (VUlnerable coDe clone DiscoverY), a scalable approach for code clone detection. This approach is specifically designed to accurately find vulnerabilities in a massive code base. To achieve the goal of highly scalable yet accurate code clone detection from a security perspective, we use the functions in a program as a unit for code clone detection. Since a function delivers both syntactic and symbolic information of the code, we are able to guarantee high accuracy in detecting clones with respect to security issues. Moreover, by applying carefully designed abstraction and normalization schemes to functions, clones with common modifications (e.g., variable names) can be detected, which in turn enables VUDDY to identify unknown vulnerabilities, as well. In addition, a clever classification of functions based on the length of a function body considerably reduces the search space, and thus enables VUDDY to work scalably even on a massive code base. With this design, VUDDY accomplishes an unprecedented balance between high scalability and accuracy.

In addition, we present a detailed explanation of the principles and implementation of VUDDY, as well as the application of the proposed approach for the detection of vulnerabilities. We further propose a method to collect CVE vulnerabilities in an automated way. From 9,770 vulnerability patches obtained from eight well-known Git repositories (e.g., Google Android), we retrieved 5,664 vulnerable functions that address 1,764 CVEs. Our evaluation involves empirically measuring the performance of VUDDY and then evaluating the practical merits of VUDDY by demonstrating the vulnerabilities detected from a pool consisting of real-world open source software. This pool includes 25,253 active C/C++ projects collected from GitHub, Linux kernels, and the Android OS of a smartphone that was released in March 2016. The results show that VUDDY preprocesses the 172 M functions (in 13.2 M files, 8.7 BLoC) of the 25,253 projects in 4 days and 7 hours, then identifies 133,812 vulnerable functions in approximately 1 second for each project. VUDDY is twice faster than ReDeBug, while having no false positive with Android firmware. Meanwhile, ReDeBug had 17.6 % false positives.

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 processes a billion lines of code in 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.
- *Automated vulnerability acquisition:* We introduce a fully automated method for acquiring known vulnerable functions from Git repositories, by taking advantage of security patch information.
- *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 11 months, 14 billion lines of code have been queried to our open service, and 144,496 vulnerable functions have been detected. Please see https://iotcube.net/

The remainder of this paper is organized as follows. Section II clarifies the taxonomy and summarizes existing approaches concerning code clone detection. Section III presents the problem and goal. Section IV describes the principles of our proposed approach, VUDDY. In Section V, we explain how VUDDY is applied to vulnerability discovery. In Section VI, we discuss issues regarding the implementation of VUDDY. Then in Section VII, we describe various experiments conducted for evaluating the scalability, time, and accuracy of VUDDY against the most competitive techniques on real-world programs. Section VIII compares VUDDY with ReDeBug, the most competitive technique, in detail. A qualitative evaluation is given through case studies in Section IX. Section X presents a discussion, and Section XI offers the conclusion and future work.

II. Taxonomy and Related Work

A. Taxonomy

To avoid confusion concerning the various taxonomies adopted in other research, we use the following well accepted ([20], [21], [22], [23]) definitions of the types of code clones.

- **Type-1: Exact clones.** These code fragments are duplicated without any change, i.e., are unmodified.

- **Type-2: Renamed clones.** These are syntactically identical clones except for the modification of types, identifiers, comments, and whitespace. VUDDY covers Type-1 and Type-2 clones.
- **Type-3: Restructured clones.** Further structural modification (e.g., deletion, insertion, or rearrangement of statements) is applied to renamed clones to produce restructured clones.
- **Type-4: Semantic clones.** These are clones that could be syntactically different, but convey the same functionality.

For the purpose of making VUDDY optimized for detecting security-related clones, we devotedly designed VUDDY to be able to detect Type-1 and Type-2 clones, which is the right scope that retains the context while allowing minor changes that frequently occur after code cloning. In the following sections, we explain why two former types of clones properly handle the security-aware context, and how approaches for detecting Type-3 and Type-4 clones sacrifice accuracy, and lead to increased false positive rate.

We also specify a granularity unit which refers to the scale of a code fragment, which is referenced throughout this study.

- **Token**: This is the minimum unit the compiler can understand. For example, in the statement `int i = 0;` five tokens exist: `int`, `i`, `=`, `0`, and `;`.
- **Line**: This represents a sequence of tokens delimited by a new-line character.
- **Function**: This is a collection of consecutive lines that perform a specific task. A standard C or C++ function consists of a header and body. A header includes a return type, function name, and parameters, and a body includes a sequence of lines that determine the behavior of the function.
- **File**: This contains a set of functions. A file may in fact contain no functions. However, most source files usually contain multiple functions.
- **Program**: This is a collection of files.

In summary, a program is a collection of files that contain functions, and a function is a collection of lines that are composed of tokens. Code cloning can occur with any of the listed granularity units. VUDDY take a function as its processing granularity.

B. Related work

Rataan et al. reviewed an extensive amount of research on code clone detection, and reported more than 70 techniques published in 11 journals and 37 conferences and workshops [23]. In this section we review some of the representative techniques which can be grouped into five categories based on the clone granularity level: set of tokens, set of lines, set of functions, files, or a hybrid of others. The selection of the granularity level greatly affects the ensuing clone detection process, and contributes greatly to scalability and accuracy.

*1) **Token-level granularity**:* Techniques that adopt token-level granularity lexically analyze a program in order to transform it into a sequence, or bag, of tokens. The token sequences are then compared for similarity comparison. The best-known of these are CCFinder [15] and CP-Miner [24]. In CCFinder, the similarity of the sequence of lexical components, i.e., tokens, is measured by a suffix-tree algorithm, which is computationally expensive and consumes a large amount of memory.

CP-Miner parses a program and compares the resulting token sequences using the "frequent subsequence mining" algorithm known as CloSpan [25]. Due to CloSpan's heuristics for improved efficiency, CP-Miner can scale to the size of moderately large code bases such as the Linux kernel, by consuming less memory space. However, the mining complexity of CP-Miner is still $\mathcal{O}(n^2)$ in the worst case where n is the number of LoC, and their experiment showed that CP-Miner requires a similar amount of execution time as CCFinder.

Aside from scalability issues, the two aforementioned techniques generate a high false positive rate caused by their aggressive abstraction, and filtering heuristics. Although the developers of CP-Miner claim that CP-Miner detects 17 to 52 percent more clones than CCFinder, Jang et al. [18] revealed that the false positive rate for reported code clones was 90 % for CP-Miner. This implies that this design does not guarantee sufficient reliability to be useful for vulnerability detection.

*2) **Line-level granularity**:* ReDeBug [18] takes a set of lines as its processing unit. It slides a window of n (4, by default) lines through the source code and applies three different hash functions to each window. The code clones between files are detected by means of membership checking in a bloom filter, which stores the hash values of each window. Although the line-based window sliding technique enables ReDeBug to detect some of the Type-3 clones, ReDeBug cannot detect Type-2 clones in which variables are renamed or data types are changed. Consequently, ReDeBug misses many vulnerable clones with slight modifications. Moreover, the use of a line-level granularity leads to a limited information of the context, and it eventually introduces many false positive cases. In terms of performance, this approach adequately scales to 30 K SourceForge projects (922 MLoC), but it required 23.3 hours to process files and build a hash database.

*3) **Function-level granularity**:* SourcererCC [19] attempts to detect Type-3 clones by using the bag-of-tokens technique. It parses all of the functions, and creates an index consisting of the bag-of-tokens of each function. Then, it infers the similarity of functions by applying an $Overlap$ function which is computed as the number of tokens shared by two functions. If the similarity between the two functions exceeds a predetermined threshold, they are deemed as a clone pair. They reduced the number of similarity computations, by using filtering heuristics that assign more weight to frequent tokens in the bag to achieve large-scale clone detection. In their experiment, SourcererCC detected code clones from 100 MLoC in approximately 37 hours, whereas it required 78 hours for CCFinder to execute for the same code base, despite CCFinder detecting fewer clones than SourcererCC. However, as a tradeoff for detecting Type-3 clones, their applicability for vulnerability detection is badly damaged. In many cases, vulnerabilities are suppressed by inserting a single if statement. However, SourcererCC cannot distinguish between patched (i.e., an if statement inserted) and unpatched (i.e., without an if statement, and thus vulnerable) code fragments.

Yamaguchi, et al. proposed a method named vulnerability extrapolation [26], and its generalized extension that exploits patterns extracted from the abstract syntax trees (ASTs) of functions to detect semantic clones [27]. When extracted ASTs are embedded in a vector space, their semantics can be identified by the latent semantic analysis technique. In particular, they perform a singular value decomposition on the embedded vectors, and obtain the structural patterns of functions. Although their method is capable of detecting Type-4 clones, they rely heavily on expensive operations, and the accuracy of detection is not precisely given in their analysis.

We must note that techniques that adopt a considerably high level of abstraction (e.g., a function into a bag-of-tokens, or into syntax trees) might be effective for detecting clones, but they are not suitable for accurately detecting vulnerable code clones, because security issues are very context sensitive.

*4) **File-level granularity**:* DECKARD [17] builds ASTs for each file, then extracts characteristic vectors from the ASTs. After clustering vectors based on their Euclidean distance, vectors that lie in proximity to one another within the Euclidean space are identified as code clones. Such tree-based approaches require extensive execution time, as the subgraph isomorphism problem is a well-known and time-consuming NP-Complete problem [28]. Furthermore, DECKARD does not guarantee sufficient scalability to handle the Debian Linux OS Distribution, according to Jang, et al. [18]. In addition, it was pointed out [29] that DECKARD has a 90 % false positive rate, which again suggests that code fragments with similar abstract trees are not necessarily clones.

FCFinder [30] removes comments, redundant whitespace, line breaks, and carriage returns, and then computes the MD5 hash value of each preprocessed file. It creates a hash table in which the hash values and corresponding files constitute keys and values, respectively. The overlapping hash values are regarded as file clones. In contrast to DECKARD, FCFinder demonstrates extensive scalability. It detected 915 K file clones from FreeBSD ports collection, which contains over 7 K software projects, and required 17.16 hours to complete its work. This extended scalability results from their design choice to adopt file-level granularity. However, for the same reason, their approach is not resilient to minor or major changes within files.

*5) **Hybrid granularity**:* Some techniques leverage a combination of various approaches. VulPecker [31] is a system for automatically checking vulnerability containment. It characterizes a vulnerability with a predefined set of features, then selects one of the existing code-similarity algorithms (e.g., [12], [18], [24]) which is optimal for the type of vulnerable code fragment. As it takes advantage of a variety of algorithms, it could detect 40 vulnerabilities which are not registered in the National Vulnerability Database (NVD). However, it required 508.11 seconds to check the existence of CVE-2014-8547 in project Libav 10.1 (0.5 MLoC), which makes it improper to be used against massive open source projects.

III. Problem and Goal Statement

A. Problem formulation

1) Clone detector: First, we define a clone detector. Formally, let $\mathbb{F}$ be the set of all functions in a program (e.g., a C program). Let $V \subseteq \mathbb{F}$, and $T \subseteq \mathbb{F}$ be a set of vulnerable functions, and a set of target functions, respectively. Then, a clone detector $\mathcal{C}$ is a function of the type:

$$\mathcal{C} : \mathbb{F} \to \{0, 1\} \tag{1}$$

which takes a program and returns 1 (i.e., vulnerable) or 0.

Definition 1 (Completeness)**.** *A clone detector $\mathcal{C}$ is complete with respect to V and T iff*

$$\forall f \in T : f \in V \Rightarrow \mathcal{C}(f) = 1. \tag{2}$$

2) Abstract clone detector: Now, we denote the set of abstract functions by $\hat{\mathbb{F}}$, which is generated from $\mathbb{F}$ by applying an abstraction function α:

$$\alpha : \mathbb{F} \to \hat{\mathbb{F}} \tag{3}$$

such that α is deterministic for $\forall f, f' \in \mathbb{F}$:

$$f = f' \Rightarrow \alpha(f) = \alpha(f'). \tag{4}$$

With $V \subseteq \mathbb{F}$ being a set of vulnerable functions, we can define the abstract clone detector $\hat{\mathcal{C}} : \mathbb{F} \to \{0, 1\}$ in terms of the abstraction function α:

$$\hat{\mathcal{C}}_V(f) \equiv \exists f' \in V : \alpha(f') = \alpha(f) \tag{5}$$

Obviously, the abstract clone detector $\hat{\mathcal{C}}$ is complete:

$$\forall f \in \mathbb{F} : f \in V \Rightarrow \hat{\mathcal{C}}(f) = 1. \tag{6}$$

B. Goals

The goals of this research are as follows:

1) Designing a clone detector $\mathcal{C}$ that satisfies completeness.
2) Designing an abstraction function α and associated abstract clone detector $\hat{\mathcal{C}}$, which is effective in detecting Type-1 and Type-2 code clones of known vulnerabilities.

With a given vulnerable function f, $\hat{\mathcal{C}}$ should detect its abstract clones f', such that $\hat{\mathcal{C}}(f) = \hat{\mathcal{C}}(f') = 1$. This implies that $\hat{\mathcal{C}}$ completely detects exact clones of vulnerability, as well as clones in which variable names, identifiers, data types, comments, and whitespace are modified.

Type-3 and Type-4 code clones have characteristics which make us exclude them from the s  of our abstract clone detector $\hat{\mathcal{C}}$. Most importantly, Type-3 and Type-4 code clones can be subjective to the loss of syntactic information which are crucial for a vulnerability to be triggered, because security vulnerabilities are often very sensitive to the constants and the order of statements. For example, vulnerability that leverages race condition in the `keyctl_read_key` function in security/keys/keyctl.c of the Linux kernel before 4.3.4 (CVE-2015-7550) was fixed by merely changing the order of statements within the function[1]. This incident implies that if the order information is lost after abstraction, the vulnerable condition is lost as well. Moreover, constants, referring to fixed values that do not change during the execution of a program, are key elements in many types of vulnerabilities. Intuitively, even a

[1]See commit b4a1b4f5047e4f54e194681125c74c0aa64d637d in the Linux kernel source tree.

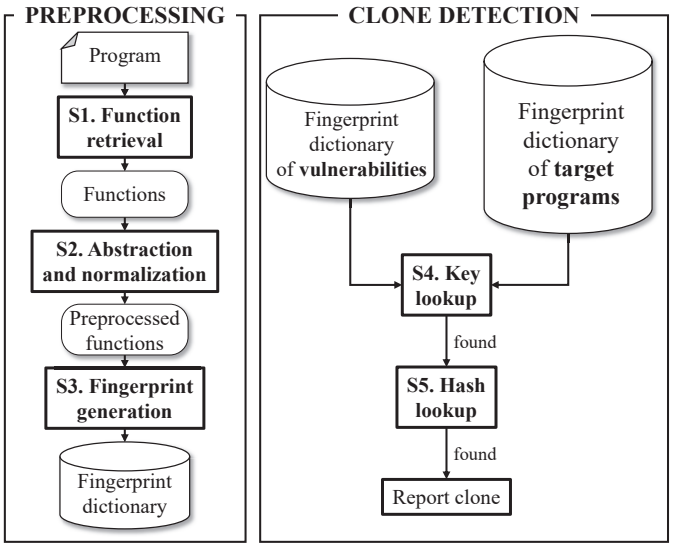


Fig. 1: Two stages of VUDDY: preprocessing and clone detection.

slight modification of the value of constants, (e.g., assigning 0 instead of 1 for a flag variable which is later used for initiating an input sanitation sequence) may suppress a vulnerability, or rather introduce a new vulnerability. For example, the denial-of-service (DoS) flaw (CVE-2012-0876) in the Expat XML parser (xmlparse.c) [32] is patched by changing the constant value 0 in the problematic function, `hash()`, into a salt variable. In that sense, if the abstraction function α does not preserve constants, it will not be able to preserve vulnerability and cause false positive.

IV. The Proposed Method: VUDDY

In this section, we describe the main ideas and principles behind VUDDY (VUlnerable coDe clone DiscoverY), which is a scalable approach to code clone detection that can be seamlessly applied to the massive OSS pool.

Based on the aforementioned goals, we propose two-stage modeling of VUDDY: preprocessing and clone detection. The preprocessing stage includes three substeps:

S1. Function retrieval
S2. Abstraction and normalization
S3. Fingerprint generation

Then in the clone detection stage, VUDDY conducts:

S4. Key lookup
S5. Hash lookup

Fig. 1 illustrates the overall stages and substeps of VUDDY. Once preprocessing is complete, the resulting fingerprint dictionary can be permanently reused, unless some portion of the program is changed. In addition, if a user specifies the changed portion, only the difference can be applied to the fingerprint in a very short time, rather than having to completely regenerate the fingerprint. This efficient design enables VUDDY to perform real-time clone detection.

A. Preprocessing

S1. Function retrieval: VUDDY 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. Blindly generating a fingerprint with an original function will lead to the omission of renamed (Type-2) clones, and cause false negatives. Thus, we establish abstraction rules to transform the function body strings before generating its fingerprint.

We fashioned four levels of abstraction which makes our approach resilient to common code modifications, while preserving vulnerability. Fig. 2 shows the transformation of a sample function at varying abstraction levels. Here, higher levels of abstraction include subordinate levels.

- **Level 0: No abstraction.** Do not abstract the code, for detecting exact (Type-1) clones.
- **Level 1: Formal parameter abstraction.** Gather formal parameters from the arguments of the function header, and replace every occurrence of the parameter variables inside the body with a symbol `FPARAM`. Then, the code clones that modify the parameter names are captured.
- **Level 2: Local variable abstraction.** Replace all local variables that appear in the body of a function with a symbol `LVAR`. From this level onwards, VUDDY is tolerant to modifications of the names of variables in copy-pasted functions, which is a common practice.
- **Level 3: Data type abstraction.** Replace the data types with a symbol `DTYPE`. The data types include not only standard C data types, and qualifiers, but also user-defined types. However, modifiers (e.g., unsigned) are not replaced because for certain types of vulnerabilities, the signedness of a variable matters. After level 3 abstraction, code clones of which the variable types have been changed (e.g., “int” to “static int”) can be detected.
- **Level 4: Function call abstraction.** Replace the name of every called function with a symbol `FUNCCALL`. Researchers have pointed out that function calls and shared APIs are typical causes of recurring vulnerabilities [12], [27], [33]. This procedure is necessary for detecting cloned functions with similar API usage.

With the abstraction scheme, VUDDY completely detects Type-2 vulnerable code clones. Cases in which abstraction is highly effective and critical for detecting vulnerable clones are introduced in section IX.

The abstracted function body is then normalized by removing the comments, whitespaces, tabs, and line feed characters, and by converting all characters into lowercase. This guarantees that the performance, especially the detection accuracy, of VUDDY is not affected by syntactically meaningless modifications. For example, if a function is inlined after cloning, or if comments in a function are greatly changed, code normalization still enables VUDDY to detect the function.

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

Level 0: No abstraction.

```
1 void avg (float arr[], int len) {
2   static float sum = 0;
3   unsigned int i;
4   for (i = 0; i < len; i++);
5     sum += arr[i];
6   printf("%f %d",sum/len,validate(sum));
7 }
```

Level 1: Formal parameter abstraction.

```
1 void avg (float FPARAM[], int FPARAM) {
2   static float sum = 0;
3   unsigned int i;
4   for (i = 0; i < FPARAM; i++)
5     sum += FPARAM[i];
6   printf("%f %d",sum/FPARAM,validate(sum);
7 }
```

Level 2: Local variable name abstraction.

```
1 void avg (float FPARAM[], int FPARAM) {
2   static float LVAR = 0;
3   unsigned int LVAR;
4   for (LVAR = 0; LVAR < FPARAM; LVAR++)
5     LVAR += FPARAM[LVAR];
6   printf("%f %d",LVAR/FPARAM,validate(LVAR));
7 }
```

Level 3: Data type abstraction.

```
1 void avg (float FPARAM[], int FPARAM) {
2   DTYPE LVAR = 0;
3   unsigned DTYPE LVAR;
4   for (LVAR = 0; LVAR < FPARAM; LVAR++)
5     LVAR += FPARAM[LVAR];
6   printf("%f %d",LVAR/FPARAM,validate(LVAR));
7 }
```

Level 4: Function call abstraction.

```
1 void avg (float FPARAM[], int FPARAM) {
2   DTYPE LVAR = 0;
3   unsigned DTYPE LVAR;
4   for (LVAR = 0; LVAR < FPARAM; LVAR)
5     LVAR += FPARAM[LVAR];
6   FUNCCALL("%f %d",LVAR/FPARAM,FUNCCALL(LVAR));
7 }
```

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

2-tuple. 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.

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 are 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. Further discussion on a proper threshold setting is provided in subsection V-A.

B. Clone detection

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

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.

```
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.

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 same integer key, then it 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.

This design of VUDDY accelerates the process of clone searching by taking advantage of the following two facts:

1) 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.

2) 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 (stable kernel released on Sep. 30, 23K files with over 15.4 MLoC) only contains 5,245 integer keys, and among the hash sets associated to the keys, the largest set has 1,019 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.

The efficiency of VUDDY in terms of discovering code clones from large programs is further evaluated in section VII.

V. Application: Vulnerability Detection

In this section we describe the application of VUDDY to detect vulnerabilities from small to massive real-world programs. To obtain vulnerable functions from reliable software projects when establishing a vulnerability database, we leveraged the Git repositories of well-known and authoritative open source projects: Google Android, Codeaurora Android Project, Google Chromium Project, FreeBSD, Linux Kernel, Ubuntu-Trusty, Apache HTTPD, and OpenSSL. Then, by using the general clone detection procedure explained in section IV, VUDDY searches for the code clones of vulnerable functions from a target program.

A. *Establishing a vulnerability database*

The process of collecting vulnerable code and establishing a vulnerability database is fully automated. The process of reconstructing vulnerable functions out of Git commit logs consists of the following steps:

1) **`git clone` repository.** This is to download specified Git repository into a local directory.
2) **`git log --grep='CVE-20'` for each repository.** This searches the commits regarding Common Vulnerability and Exposures (CVEs) [34]. This works for any general keywords, such as vulnerability types, or vulnerability names. If it is required to collect certain types of bugs, such as buffer overflow, the keyword for grep would be "buffer overflow." Well-known vulnerability names, such as Heartbleed, can also be queried.
3) **`git show` the searched commits.** This shows the full commit log that contains a description of the vulnerability related to CVE, as well as a security patch information in unified diff format. Diffs have a dedicated line for recording file metadata, in which reference IDs to old and new files addressed by the patch are written.
4) **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 put all the messages of associated commits together are 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.
5) **`git show` the old file ID.** This shows the old, unpatched version of the file. We then retrieve the vulnerable function from the file.

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 indicate 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 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 +   account_pipe_buffers(pipe, pipe->buffers, 0);
9 +   free_uid(pipe->user);
10    for (i = 0; i < pipe->buffers; i++) {
11       struct pipe_buffer *buf = pipe->bufs + i;
12       if (buf->ops)
```

Listing 2: Snippet of the vulnerable function retrieved from the patch for CVE-2013-4312.

```
1 void free_pipe_info(struct pipe_inode_info *pipe)
2 {
3     int i;
4     for (i = 0; i < pipe->buffers; i++) {
5        struct pipe_buffer *buf = pipe->bufs + i;
6        if (buf->ops)
```

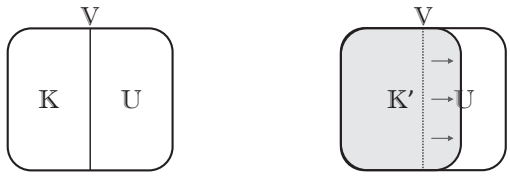
Applying the same method to 9,770 vulnerability patches, we collected 5,664 vulnerable functions that address 1,764 unique CVEs. These vulnerable functions have well-known vulnerabilities such as buffer overflow, integer overflow, input validation error, permission-related vulnerabilities, and others. The shortest vulnerable function consists of 51 characters after abstraction and normalization. Single-lined functions (e.g., a guard function which returns by calling another function) are excluded from the database, since these functions frequently cause false positives when our abstraction is applied.

B. *Vulnerability 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 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 $\mathcal{C}$ only considers exact (Type-1) clones, then the coverage of $\mathcal{C}$ is $\mathbb{K}$. However, by the use of our abstraction strategy, the coverage of an abstract clone detector $\hat{\mathcal{C}}$ 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 code clones discovered by VUDDY. The examples are provided in subsection IX-C.



(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.

VI. Implementation

We implemented VUDDY[2] in Python 2.7.11, and the robust parser with the ANTLR parser generator 4.5.3 [35]. In this section, we discuss issues related to the implementation.

A. Generating a robust parser

One intuitive approach for obtaining functions from a program and analyzing their syntax is to use a compiler. However, the use of parsers integrated in compilers is evidently restricted to the occasions when a working build environment is available. In addition, even if we succeed to replicate the working environment, the source code may not be complete or may contain syntax errors, which blights the whole parsing procedure [36], [37]. Thus, we ensure that our method is feasible and sufficiently general to be used in practice by generating and utilizing a robust parser for C/C++ based on the concept of fuzzy parsing with the utilization of island grammars [38], [39]. This parser does not require a build environment or header information, which means it is able to parse an individual file, and does not fail when it encounters syntactic errors during parsing. Even when a broken or partial code is given, it parses as much as it understands.

B. Selection of hash function

Any hash function can be used for fingerprint generation. However, we impose three constraints that need to be considered in order to maximize the scalability and speed of our approach. First, to prevent two or more different and irrelevant functions from having the same hash value, it is necessary to avoid hash collision as much as possible. Second, building the smallest possible fingerprint dictionary requires us to choose a hash function that produces the fewest possible bits of hash values. Third, to minimize the hashing time, the chosen function needs to be fast, and its implementation well-optimized. For VUDDY, we selected the MD5 hash algorithm, which completed hashing 20 million randomly generated alphanumeric strings with their size ranging from 51 to 1,000 bytes, in only 15 seconds without collision. Non-cryptographic hash algorithms such as CityHash [40], MurmurHash [41], and SpookyHash [42] were also considered, but they required a similar amount of time for the experiment. Thus, we decided to adopt MD5 which is provided as a built-in method of the Python Hashlib package, rather than taking unidentified risks by using third party hash libraries. Although we are aware that the MD5 hash algorithm suffers from cryptographic weaknesses [43], two facts make MD5 sufficient: Our use of MD5 is not for cryptography; and VUDDY is designed such that hash collision occurs only when two different functions have identical lengths. Note that we exclude the use of fuzzy hash algorithms, which produce similar hash values for similar plaintexts, as distinguishing slightly modified clones from semantically changed code presents another problem.

C. Dictionary implementation

The fingerprint dictionary is a crucial data structure in the implementation of VUDDY because it dramatically reduces the search space of possible clones and thus expedites the whole process. As previously stated, a dictionary is an associative container that maps keys to values. We chose to use the built-in dictionary data structure of Python, with which the average time complexity is $O(1)$ for the `in` operation to check the existence of an element among the keys of a dictionary, regardless of the number of elements.

VII. Evaluation

We proceed to evaluate the efficacy and effectiveness of VUDDY in two aspects: scalability and accuracy, by comparing VUDDY with other state-of-the-art techniques.

A. 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 during the period from January 1st, 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

[2]Our implementation is available at https://iotcube.net/

TABLE I: Scalability and time comparison for varying input size. The average time was computed after iterating each experiment five times.

LoC	VUDDY	SourcererCC	ReDeBug	CCFinderX	DECKARD
1 K	**0.44 s**	2.3 s	35.6 s	6 s	1 s
10 K	**0.81 s**	3.1 s	35.6 s	10 s	3 s
100 K	**5.17 s**	50.7 s	42 s	50 s	13 s
1 M	**55 s**	1 m 44 s	1 m 43 s	6 m 44 s	2 m 20 s
10 M	**12 m 43 s**	24 m 38 s	18 m 32 s	1 h 36 m	12h 30 m
100 M	**1 h 32 m**	9 h 42 m	2 h 32 m	12 h 44 m	**Memory ERROR**
1 B	**14 h 17 m**	**25 d 3 h**	**1 d 3 h**	**File I/O ERROR**	–

TABLE II: Configurations for experiments.

Technique	Configuration
SourcererCC	Min length 6 lines, min similarity 70 %.
ReDeBug	n-gram size 4, 10 context lines.
DECKARD	Min length 50 tokens, similarity 85 %, 2 token stride.
CCFinderX	Min length 50 tokens, min token types 12.

to the Github projects, we downloaded the firmware of several Android smartphones.

B. Scalability evaluation

First, we evaluated the scalability of VUDDY, against four publicly available and competitive techniques (SourcererCC, ReDeBug, DECKARD, and CCFinderX) in terms of varying target program size. Note that as Wang et al. [44] pointed out, the configuration choices have a significant impact on the behavior of the tools that are compared. As a remedy, we referenced the optimal configuration of each technique found by [44], [45], and [19] to conduct a sufficiently fair evaluation. The configuration can be found in Table II.

To focus on 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 I, VUDDY overwhelmed other techniques. DECKARD had the least scalability, failing to process 100 MLoC target because of a memory error. In the case of CCFinderX, a file I/O error occurred after 3 days of execution for a 1 BLoC target. VUDDY finished generating fingerprints and detecting clones of the 1 BLoC target in only 14 hour and 17 minutes. Although SourcererCC and ReDeBug 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. Fig. 6 displays a graph depicting the results in Table I. We can clearly see that the execution time of the other state-of-the-art techniques explodes as the target size grows. In fact, VUDDY scales even to the size of all 25,253 repositories consisting of 8.7 BLoC with ease, requiring only 4 days and 7 hours.

C. 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. In this subsection,

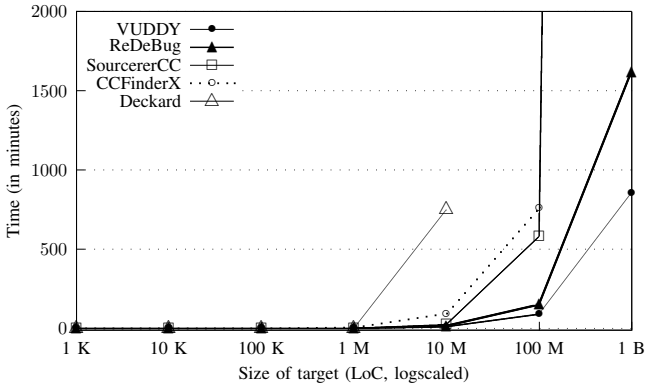


Fig. 6: Execution time when varying size of target programs were given to VUDDY, SourcererCC, ReDeBug, DECKARD, and CCFinderX. DECKARD and CCFinderX scaled up to 10 MLoC and 100 MLoC, respectively, then failed to execute. Although ReDeBug and SourcererCC scales to 1 BLoC, their growth rates of time are much steeper than that of VUDDY.

we focus on comparing the accuracy of VUDDY and other clone detection techniques: SourcererCC, DECKARD, and CCFinderX. These techniques are not aimed at detecting "vulnerable" clones, and thus are not accurate when finding security vulnerabilities. On the other hand, ReDeBug is designed for detecting vulnerable code clones. Thus, we compare VUDDY with ReDeBug in detail in section VIII.

To evaluate accuracy on the most equitable basis possible, we decided to conduct clone detection using each technique, then manually inspect every reported clones. The result of clone detection between our vulnerability database and Apache HTTPD 2.4.23 (352 KLoC) is shown in Table III. 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 III only accounts for 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. We reused the configurations of Table II except for the minimum similarity threshold settings.

VUDDY reported 9 code clones in 22 seconds, and all of the findings were unpatched vulnerable clones in Apache HTTPD 2.4.23. 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

TABLE III: Accuracy of VUDDY, SourcererCC, DECKARD, and CCFinderX when detecting clones between the vulnerability database and Apache HTTPD 2.4.23.

Technique	Time	Rep†	TP	FP	FN	Precision
VUDDY	22 s	9	9	0	2	1.000
SourcererCC (100)*	122 s	1	1	0	8	1.000
SourcererCC (70)*	125 s	56	2	54	7	0.036
DECKARD (100)*	58 s	57	3	54	8	0.053
DECKARD (85)*	234 s	462	4	458	8	0.009
CCFinderX	1201 s	74	11	63	1	0.147

* The values between parentheses denote minimum similarity threshold configuration in percent

† Denotes the number of clones each technique reported

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. 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 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. [29] which claims that DECKARD has 90 % false positive. CCFinderX was the only technique that reported more true positive cases than those of VUDDY. 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 quite 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.

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 (see Equation 2).

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);
```

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.

D. 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.

VIII. In-depth comparison with ReDeBug

In subsection VII-C, we compared VUDDY with other clone detection techniques of which the designs do not consider vulnerability preservation in clones. Here, we compare VUDDY with ReDeBug, a line-based vulnerability detection technique which takes advantage of security patches to find vulnerability. The sole purpose of ReDeBug is to scalably find vulnerable code clones, which we believe to be very similar to our purpose. As ReDeBug takes a different design choice, (i.e., line-level granularity) for clone detection, we compare VUDDY and ReDeBug in detail to demonstrate the efficiency and effectiveness of VUDDY. Three major advantages of VUDDY over ReDeBug are as follows:

- VUDDY is twice faster than ReDeBug.
- VUDDY has far less false positive than ReDebug.
- VUDDY is capable of detecting Type-2 clones, but ReDeBug is not.

TABLE IV: Comparison of VUDDY and ReDeBug, targeting Android firmware (14.86 MLoC, 349 K functions).

	VUDDY	ReDeBug
Time to complete	17 m 3 s	28 m 15 s
# initial reports	206	2,090
# multiple counts	0	1,845
# unique clones	206	245
# false positives	0	43
# end result	206	202
# unique findings	25	21
# common findings	181	

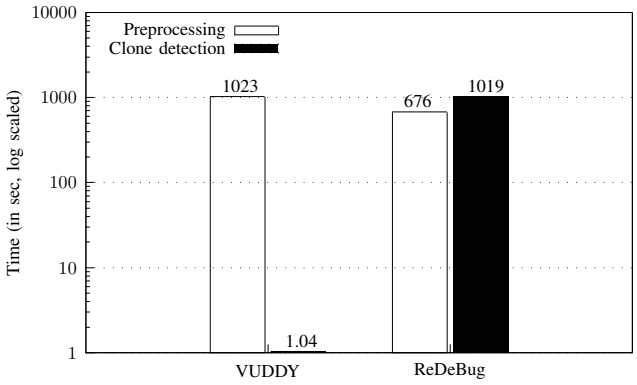


Fig. 7: Time required for preprocessing and clone detection.

A. Speed

The result of comparison is summarized in Table IV. In terms of speed, VUDDY is faster than ReDeBug. When querying 9,770 CVE patches targeting Android firmware (14.9 MLoC, using kernel version 3.18.14), VUDDY required 17 minutes and 3 seconds. Meanwhile, ReDeBug using the same default parameters ($n = 4$, $c = 3$, where n is the number of lines per window, and c is the amount of context) used by Jang et al. [18] in their experiment, required 28 minutes and 15 seconds for clone detection. In fact, VUDDY required 1,023 seconds for preprocessing and for the fingerprint generation procedure, and the actual clone detection required only 1.04 seconds, as illustrated in Fig. 7. Note that once the preprocessing is complete, VUDDY does not need to regenerate the fingerprint dictionary for every clone detection. Thus, we can argue that VUDDY detects vulnerable code clones at a speed more than twice faster than ReDeBug, in practice.

B. False positive

VUDDY overwhelms ReDeBug with decisive margin, with respect to accuracy. No false positive was reported by VUDDY. However, we conducted a manual inspection for 12 hours with 2,090 code clones reported by ReDeBug to find that 1,845 (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 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 code 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);
```

C. False negative

Table IV 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.

```
1   ssid = (wlc_ssid_t *) data;
2   memset(profile->ssid.SSID, 0,
3     sizeof(profile->ssid.SSID));
4 + profile->ssid.SSID_len = MIN(ssid->SSID_len,
      DOT11_MAX_SSID_LEN);
5   memcpy(profile->ssid.SSID, ssid->SSID, ssid->
        SSID_len);
6   profile->ssid.SSID_len = ssid->SSID_len;
7   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 precisely the same reason addressed in subsection VII-C. 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.

IX. Case Study

Taking advantage of the scalability and accuracy of VUDDY, we could investigate a wide range of programs in a relatively short period of time. In this section, we evaluate the practical merits of VUDDY by demonstrating vulnerabilities detected in real-world programs.

According to the scale and cause of clones, we classify clone-induced vulnerabilities into the following three categories: Library reuse; Kernel reuse; and Intra-project code reuse cases. The cases we introduce show that software is often affected by old vulnerabilities blended into their system by code cloning.

A. Library reuse cases

In practice, library reuse takes place very frequently, because any software, small or large, can use libraries without much restriction. For example, in the latest release of VLC media player, an open source media player, at least 91 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.

1) LibPNG and a mobile browser: The smartphone we addressed in subsection IX-B is shipped with a built-in web browser application based on the use of an outdated LibPNG library. The version of LibPNG in the web browser is 1.2.45, which was released in July 2011. VUDDY detected that the fix for CVE-2011-3048 (Heap-based buffer overflow in LibPNG) is not applied in that version, leaving the browser vulnerable. In Listing 8, the vulnerable function named `png_set_text_2` in `pngset.c` is described.

Listing 8: Snippet of vulnerable function in pngset.c of LibPNG 1.2.45.

```
1  if (info_ptr->text != NULL)
2  {
3  png_textp old_text;
4  int old_max;
5  old_max = info_ptr->max_text;
6  info_ptr->max_text = info_ptr->num_text + num_text +
       8;
7  old_text = info_ptr->text;
8  info_ptr->text = (png_textp)png_malloc_warn(png_ptr,
       (png_uint_32)(info_ptr->max_text * png_sizeof(
     png_text)));
9  if (info_ptr->text == NULL)
10 {
11 png_free(png_ptr, old_text);
12 return(1);
13 }
14 png_memcpy(info_ptr->text, old_text, (png_size_t)(
     old_max * png_sizeof(png_text)));
15 png_free(png_ptr, old_text);
16 }
```

When memory allocation in line 8 fails, `png_free` function is called without restoring the prior states of `info_ptr->max_text`, and `info_ptr->old_text`. As a result, dangling pointers are generated. We could exploit this vulnerability to accomplish denial of service through a crafted PNG file.

This case is very alarming because a vulnerability which had already been patched five years ago is still being distributed through widely-used smartphones. After we reported this bug to the manufacturers, they affirmed that they will conduct a dependency check and library update for the next release.

2) Expat library and Apache HTTPD: In the Apache HTTP server, we discovered a vulnerable code clone of CVE-2012-0876, which eventually turned out to be a zero-day vulnerability. The latest stable release (2.4.23), and a few recent releases (2.4.18 and 2.4.20) are affected. Apache HTTP server relies on the Expat library for parsing XML files. Unfortunately, the library that is currently being used by Apache HTTP server is an outdated version, which is vulnerable to CVE-2012-0876, a so-called Hash DoS attack. VUDDY detected

that `expat/lib/xmlparse.c` contained a code clone of a vulnerable function retrieved from Google Android repository, which allows attackers to cause a DoS attack through a crafted XML file. 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.

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. 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.

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: Vulnerable function in httpd-2.4.23/srclib/apr-util/xml/expat/lib/xmlparse.c (lines from 5428 to 5434) of Apache HTTP server 2.4.23 which is still unpatched even though the security patch was released in 2012.

```
...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;...
```

B. Kernel reuse cases

One important characteristic of cases in which the kernel is reused, is that these kernels usually lag behind the latest kernel. This is very prevalent in the ecology of IoT devices including Android smartphones, Tizen appliances, and Linux-oriented operating systems such as Ubuntu. It often takes at least half a year to develop their own operating system on the basis of a certain version of Linux kernel, which eventually leaves them (i.e., the IoT devices, OS distributions, and smart appliances running the Tizen OS) subject to the vulnerabilities reported during the period of development. In other words, although the Linux kernel is constantly patched and updated, devices inevitably lag behind the patching efforts of Linux kernel developers.

The dirty COW vulnerability (CVE-2016-5195) is an excellent example of a situation in which VUDDY can be effective, because a vulnerable Linux kernel was reused in a smartphone. This 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 (see Listing 11) in the firmware of an Android smartphone released in March 2016, and we successfully exploited the vulnerable clone to gain root privilege of the smartphone running the examined firmware. 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 those in Android smartphones which hold more than half of the market share.

Listing 11: Vulnerable clone affected by Dirty COW vulnerability found in the Android firmware.

```
...}
   if ((flags & FOLL_NUMA) && pte_protnone(pte))
      goto no_page;
   if ((flags & FOLL_WRITE) && !pte_write(pte)) {
      pte_unmap_unlock(ptep, ptl);
      return NULL;
   }...
    * reCOWed by userspace write).
    */
   if ((ret & VM_FAULT_WRITE) && !(vma->vm_flags &
       VM_WRITE))
      *flags &= ~FOLL_WRITE;
   return 0;
}
```

We contacted the smartphone manufacturer[3] to inform them about our findings of the vulnerability. As a result, they immediately initiated a process to address the vulnerability.

C. Intra-project code reuse 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 trees including versions 4.8.6, 4.7.10, and the LTS versions of the 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 patched in 2008 (Listing 12), the nilfs2 file system of which the implementation is very similar (but differs in relation to some identifiers) to that of ext2 has remained unpatched to date. The problematic function named `*nilfs_dotdot` is in `linux/fs/nilfs2/dir.c`.

Listing 12: 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 13: 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 13 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 13 is different from that of the original buggy

[3]Name of this company and vulnerable smartphones are deliberately anonymized for legal reasons.

function (`ext2_get_page`), this is detected by VUDDY because as described in subsection IV-A-S2, 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.

X. Discussion

A. The use of function-level granularity

In this section, we discuss the reasoning behind selecting the function level as the basis for clone detection, through a theoretical and empirical analysis of time complexity, storage use, and accuracy of detection. As a way of answering the research question of "Which granularity is best for scalable and accurate vulnerability detection?" we parameterize the granularity level, and observe the performance curve of VUDDY.

1) Time complexity: Given a function F consisting of l LoC, c characters per line on average, assume that we use g lines as a granularity unit. Then, each processing window will consist of g blocks, and the number of windows in F will be $l - g + 1$. We can compute the cost of preprocessing a function as the multiplication of the number of windows by the preprocessing time per window. When preprocessing a retrieved function, VUDDY conducts abstraction, normalization, and length computation, and then applies the MD5 hash algorithm. However, the first three operations require a trivial amount of time relative to the MD5 hash computation, of which the complexity is denoted as $ax + b$ where x is the number of input characters[4]. Thus, the $Cost$ function is approximated as follows:

$$Cost(g) = (\#windows) * (HashTime/window) \quad (7)$$
$$= (l - g + 1) * (a(cg) + b) \quad (8)$$
$$= -acg^2 + (acl + ac - b)g + b(l + 1) \quad (9)$$

This formula is empirically validated by measuring the time VUDDY requires to preprocess functions of various lines of code. As illustrated in Fig. 8, the cost function is parabolic, peaking at $g = 2/l$. The minimum value of each graph is attained when $g = l$, which means the least time is required for any function when we take advantage of the function-level granularity. Note that these properties are observed for all functions in our vulnerability database, with sizes ranging from 51 to 2526, but in consideration of the readability of the graphs, we only plotted six representative functions.

[4]To the best of our knowledge, the time complexity of MD5 implementation in Python Hashlib is not known. Therefore, we empirically measured the time complexity in subsection VI-B to find that it is linear to the length of input plaintext.

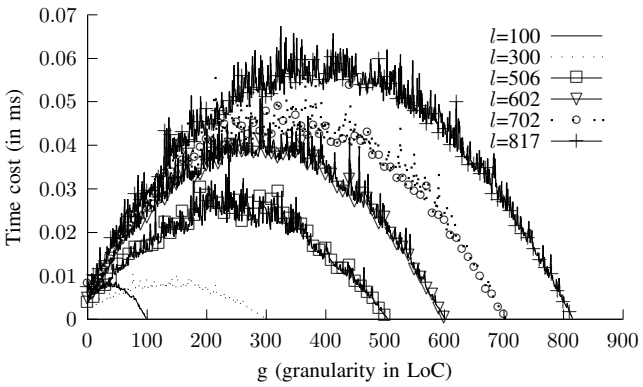


Fig. 8: Time required for preprocessing the functions by varying the granularity level. The graphs represent six functions with 100, 300, 506, 602, 702, and 817 LoC, respectively.

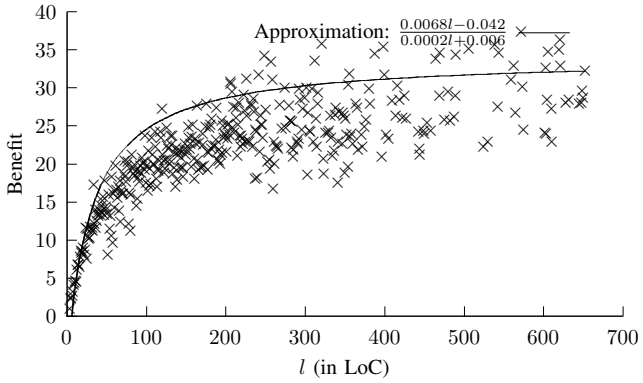


Fig. 9: Relative time benefit of $cost(g = l)$ (i.e., function-level granularity) over $cost(g = 4)$ (i.e., as of ReDeBug) along varying sized functions in the vulnerability database. Each point (x-shaped) represents a function consisting of l lines, and the curved line represents an approximate fitted curve.

Using the cost function, we now estimate the relative benefit, (i.e., speedup) of function-level granularity (VUDDY) over using four LoC as granularity (default of ReDeBug). The $Benefit$ function is given by:

$$Benefit(l) = \frac{cost(g = 4)}{cost(g = l)} \quad (10)$$
$$= \frac{(4ac + b)l - (12ac + 3b)}{acl + b} \quad (11)$$

To prove the validity of our theoretical estimation, we conducted another experiment to assess the benefit, with the functions of our vulnerability database. The result is shown in Fig. 9. By observing the open source projects in our dataset, we determined the average number of characters, c, to be 10, and through the experiment in subsection VI-B, the complexity of the MD5 hash algorithm was found to be $2.00e^{-5}x + 0.006$. This observation, enabled us to obtain a graph that approximately fits the experimental data (the curved graph in Fig. 9), where $Benefit(l)$ asymptotically approaches 34 as l increases.

2) Memory usage: Memory usage is another crucial factor that determines the scalability of a method. Technically, our

method takes variable granularity l, which always equals the length of a function. This is highly advantageous compared to approaches that utilize a fixed granularity-level, e.g., ReDeBug or CCFinder, in terms of memory usage. Table V shows the amount of memory used when the functions in our vulnerability database are preprocessed. When l was taken as the granularity level, the least memory was used because only one fingerprint was generated per function. When the granularity g was 4 LoC, it consumed 651 MB of memory for processing and storing the fingerprints of fine-grained functions. This is considerable overhead, considering that the total size of files in the vulnerability database is 21 MB (see subsection V-A).

Formally, $l - g + 1$ fingerprints were generated for each function, which led an increase in the amount of memory space required. It can be observed in Table V that when the granularity level is increased, memory use decreases. However, this does not necessarily imply that the use of larger granularity reduces memory usage. Instead, the main reason for the lower memory usage is that functions shorter than the fixed granularity are not processed. For example, if we take 100 LoC as the granularity level, functions with a length is shorter than 100 cannot be processed, and are thus discarded. Therefore, we can confidently argue that function-level granularity promotes memory efficiency more than other fixed granularity levels.

3) Accuracy: Considering the former discussion from a different standpoint, fixed granularity can impair the accuracy of an approach. In other words, an approach that attempts to detect vulnerable code clones using granularity g cannot detect clones with a length smaller than g, which causes false negatives. As a remedy, we could generate fingerprints for every possible g value and search for the corresponding fingerprint from every dictionary whenever it fails to detect functions smaller than the fixed granularity, which causes a tremendous amount of overhead. Moreover, it is obvious that the false positive rate increases when we use finer granularity, as shown in subsection VII-C. The false positive rate is directly related to the trustworthiness of a vulnerable clone detector. Vulnerable clone detectors that report numerous false alarms do not actually help developers identify the problems in their code; instead, they lead to increased efforts to routinely check meaningless alarms. In this respect, we believe that our design of VUDDY achieves the right balance between accuracy and scalability.

TABLE V: Memory use when preprocessing functions in the vulnerability database with the given granularity. l refers to the LoC of a function.

Property	Granularity (LoC)	Memory use (MB)
Variable	l	48
Fixed	4	651
	10	496
	40	172
	100	49

B. Room for speedup

Currently, a large portion of the overhead is concentrated in the parsing step. When generating a fingerprint of Android firmware, VUDDY spent 973 seconds (95.1 %) out of 1023 seconds only for parsing. To resolve this performance bottleneck, we also implemented a faster version of VUDDY, called VUDDY-fast, which utilizes regular expression to identify and analyze functions. VUDDY-fast required only 1 hour and 17 minutes for generating fingerprint dictionary of 1 BLoC input in Table I, but it identified 9.7 % less functions. As one of the future works, we plan to improve the performance of parser by optimizing the grammar and leveraging parser generator which is faster than ANTLR.

C. Open service

In April 2016, we launched an open web service via web with which anyone can use VUDDY to inspect their programs. A number of open source developers, device manufacturers, and commercial product developers tested more than 14 BLoC for 11 months, and VUDDY detected 144,496 vulnerable functions. For lack of space, the results and insights drawn from the service is described in the Appendix.

XI. Conclusion and Future Work

In this paper, we proposed VUDDY, which is an approach for scalable and accurate vulnerable code clone discovery. The design principles of VUDDY are directed towards 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.

Our work can be extended in multiple directions. Firstly, we plan to continue improving the performance of VUDDY by refining the parser and expanding the vulnerability database. It will boost the speed of VUDDY and increase detection rate. Moreover, we will try to combine our approach with other types of vulnerability-detecting techniques (e.g., fuzzers), which will allow a more sophisticated detection of vulnerability.

Acknowledgment

We thank the anonymous reviewers for their insightful comments to improve the quality of the paper. We are also grateful to Donghyeok Kim for his contribution on the parser generation, and Taebeom Kim for his help on the manual inspection process. 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).

REFERENCES

[1] M. W. Godfrey and Q. Tu, "Evolution in open source software: A case study," in *Software Maintenance, 2000. Proceedings. International Conference on*. IEEE, 2000, pp. 131–142.

[2] G. Succi, J. Paulson, and A. Eberlein, "Preliminary results from an empirical study on the growth of open source and commercial software products," in *EDSER-3 Workshop*. Citeseer, 2001, pp. 14–15.

[3] W. Scacchi, "Understanding open source software evolution," *Software Evolution and Feedback: Theory and Practice*, vol. 9, pp. 181–205, 2006.

[4] "SourceForge," http://sourceforge.net, accessed: 2016-11-01.

[5] "GitHub," http://github.com, accessed: 2016-11-01.

[6] M. Kim, L. Bergman, T. Lau, and D. Notkin, "An ethnographic study of copy and paste programming practices in OOPL," in *Empirical Software Engineering, 2004. ISESE'04. Proceedings. 2004 International Symposium on*. IEEE, 2004, pp. 83–92.

[7] C. J. Kapser and M. W. Godfrey, ""Cloning considered harmful" considered harmful: patterns of cloning in software," *Empirical Software Engineering*, vol. 13, no. 6, pp. 645–692, 2008.

[8] J. Mayrand, C. Leblanc, and E. M. Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *Software Maintenance 1996, Proceedings., International Conference on*. IEEE, 1996, pp. 244–253.

[9] M. Kim, V. Sazawal, D. Notkin, and G. Murphy, "An empirical study of code clone genealogies," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 5. ACM, 2005, pp. 187–196.

[10] T. Lavoie, M. Eilers-Smith, and E. Merlo, "Challenging cloning related problems with gpu-based algorithms," in *Proceedings of the 4th International Workshop on Software Clones*. ACM, 2010, pp. 25–32.

[11] B. S. Baker, "On finding duplication and near-duplication in large software systems," in *Reverse Engineering, 1995., Proceedings of 2nd Working Conference on*. IEEE, 1995, pp. 86–95.

[12] N. H. Pham, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Detection of recurring software vulnerabilities," in *Proceedings of the IEEE/ACM international conference on Automated software engineering*. ACM, 2010, pp. 447–456.

[13] H. Li, H. Kwon, J. Kwon, and H. Lee, "CLORIFI: software vulnerability discovery using code clone verification," *Concurrency and Computation: Practice and Experience*, pp. 1900–1917, 2015.

[14] 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 *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 692–708.

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

[16] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier, "Clone detection using abstract syntax trees," in *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, 1998, pp. 368–377.

[17] L. Jiang, G. Misherghi, Z. Su, and S. Glondu, "DECKARD: Scalable and accurate tree-based detection of code clones," in *Proceedings of the 29th international conference on Software Engineering*. IEEE Computer Society, 2007, pp. 96–105.

[18] J. Jang, A. Agrawal, and D. Brumley, "ReDeBug: finding unpatched code clones in entire os distributions," in *Security and Privacy (SP), 2012 IEEE Symposium on*. IEEE, 2012, pp. 48–62.

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

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

[21] R. Koschke, "Survey of research on software clones," in *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2007.

[22] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470–495, 2009.

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

[24] Z. Li, S. Lu, S. Myagmar, and Y. Zhou, "CP-Miner: Finding copy-paste and related bugs in large-scale software code," *Software Engineering, IEEE Transactions on*, vol. 32, no. 3, pp. 176–192, 2006.

[25] X. Yan, J. Han, and R. Afshar, "CloSpan: Mining: Closed sequential patterns in large datasets," in *Proceedings of the 2003 SIAM International Conference on Data Mining*. SIAM, 2003, pp. 166–177.

[26] F. Yamaguchi, F. Lindner, and K. Rieck, "Vulnerability extrapolation: assisted discovery of vulnerabilities using machine learning," in *Proceedings of the 5th USENIX conference on Offensive technologies*. USENIX Association, 2011, pp. 13–13.

[27] F. Yamaguchi, M. Lottmann, and K. Rieck, "Generalized vulnerability extrapolation using abstract syntax trees," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 359–368.

[28] R. C. Read and D. G. Corneil, "The graph isomorphism disease," *Journal of Graph Theory*, vol. 1, no. 4, pp. 339–363, 1977.

[29] L. Jiang, Z. Su, and E. Chiu, "Context-based detection of clone-related bugs," in *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2007, pp. 55–64.

[30] Y. Sasaki, T. Yamamoto, Y. Hayase, and K. Inoue, "Finding file clones in FreeBSD ports collection," in *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*. IEEE, 2010, pp. 102–105.

[31] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "VulPecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 2016, pp. 201–213.

[32] "The Expat XML Parser," http://expat.sourceforge.net/, accessed: 2016-11-01.

[33] M. Zhang, Y. Duan, H. Yin, and Z. Zhao, "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, pp. 1105–1116.

[34] "Common Vulnerability Enumeration," http://cve.mitre.org, accessed: 2016-11-01.

[35] "ANTLR, ANother Tool for Language Recognition," http://www.antlr.org/, accessed: 2016-11-01.

[36] N. Synytskyy, J. R. Cordy, and T. R. Dean, "Robust multilingual parsing using island grammars," in *Proceedings of the 2003 conference of the Centre for Advanced Studies on Collaborative research*. IBM Press, 2003, pp. 266–278.

[37] S. Neuhaus, T. Zimmermann, C. Holler, and A. Zeller, "Predicting vulnerable software components," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 529–540.

[38] A. Van Deursen and T. Kuipers, "Building documentation generators," in *Software Maintenance, 1999.(ICSM'99) Proceedings. IEEE International Conference on*. IEEE, 1999, pp. 40–49.

[39] L. Moonen, "Generating robust parsers using island grammars," in *Reverse Engineering, 2001. Proceedings. Eighth Working Conference on*. IEEE, 2001, pp. 13–22.

[40] G. Pike and J. Alakuijala, "Introducing cityhash," 2011.

[41] A. Appleby, "Murmurhash 2.0," 2008.

[42] B. Jenkins, "SpookyHash: a 128-bit non-cryptographic hash (2010)," 2014.

[43] X. Wang and H. Yu, "How to break MD5 and other hash functions," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2005, pp. 19–35.

[44] T. Wang, M. Harman, Y. Jia, and J. Krinke, "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, pp. 455–465.

[45] J. Svajlenko and C. K. Roy, "Evaluating modern clone detection tools," in *Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on*. IEEE, 2014, pp. 321–330.

APPENDIX
VUDDY AS AN OPEN SERVICE

VUDDY has been serviced online (at IoTcube, https://iotcube.net) 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. Here, we present the working example of VUDDY given the firmware of an Android smartphone. When the fingerprint is uploaded to our service platform, the platform shows the number of detected vulnerable clones, the origins of clones, yearly distribution of CVEs assigned to vulnerabilities, CVSS (Common Vulnerability Scoring System) score distribution, CWE (Common Weakness Enumeration) distribution, and a tree view with which users are able to locate the files affected by vulnerable clones. Graphs in the appendix are downloaded as vector images from IoTcube, and the other figures are screen-captured.

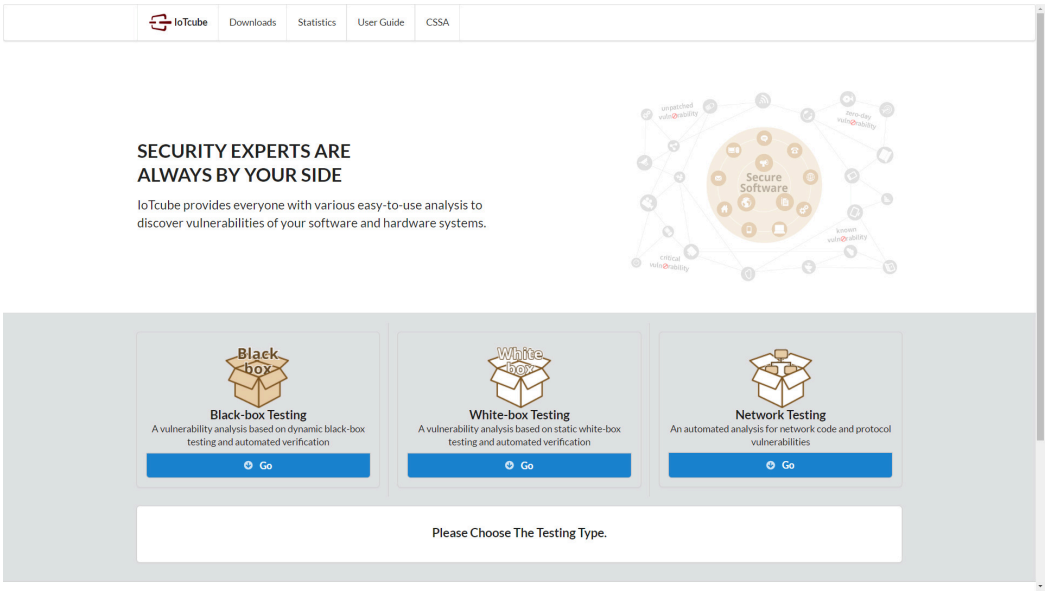


Fig. 10: The main page of IoTcube. The implementation of VUDDY is under the White-box Testing menu.

Rank	CVE	#
1	CVE-2009-0029	1,377
2	CVE-2016-1575	703
3	CVE-2010-2939	700
4	CVE-2015-2695	676
5	CVE-2006-2313	642
6	CVE-2015-3194	634
7	CVE-2015-5157	618
8	CVE-2016-5424	534
9	CVE-2015-7973	531
10	CVE-2014-3534	503

Rank	CWE	#
1	CWE-264	9,305
2	CWE-119	8,850
3	CWE-399	7,134
4	CWE-020	6,751
5	CWE-018	2,602
6	CWE-362	2,459
7	CWE-189	2,454
8	CWE-200	2,064
9	CWE-310	2,052
10	CWE-017	636

Fig. 11: Statistical knowledge obtained by web service for 11 months. Tables show the most frequently detected CVEs and CWEs, respectively. This information is also open to the users.

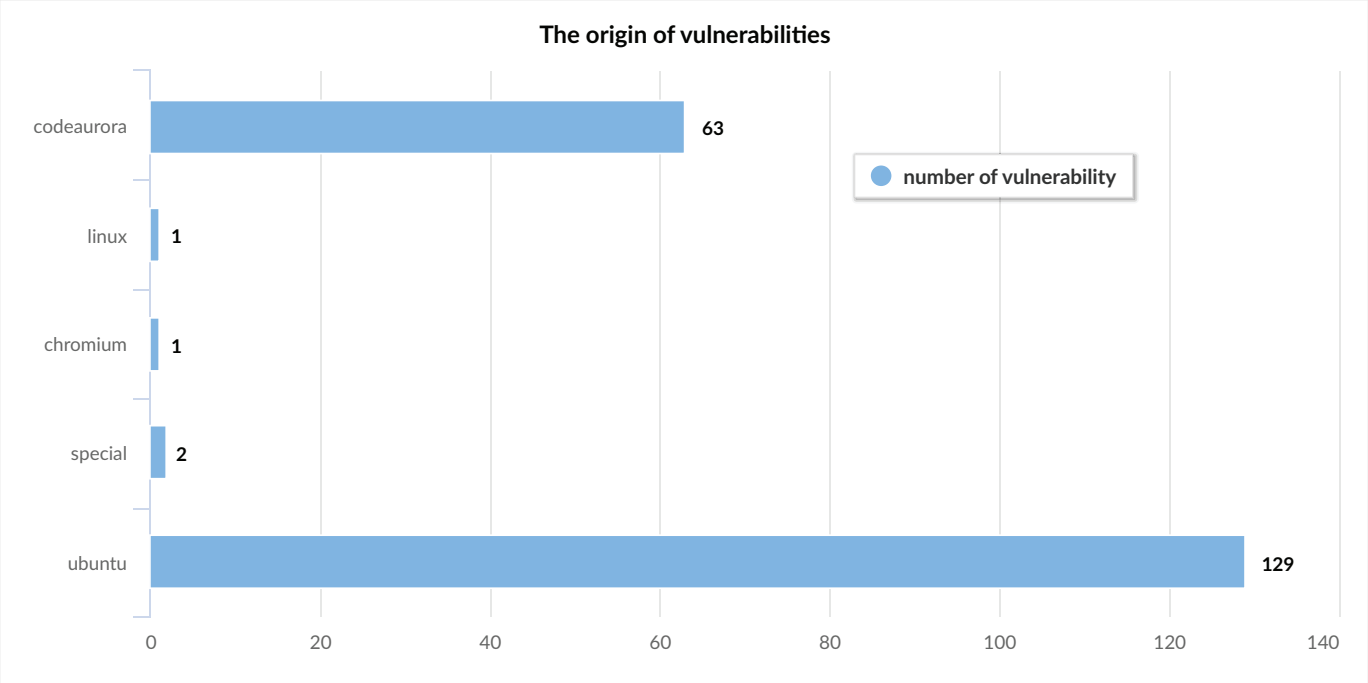


Fig. 12: The origin of vulnerabilities. 129 vulnerable functions detected in the smartphone are already reported and patched in the repository of Ubuntu-Trusty.

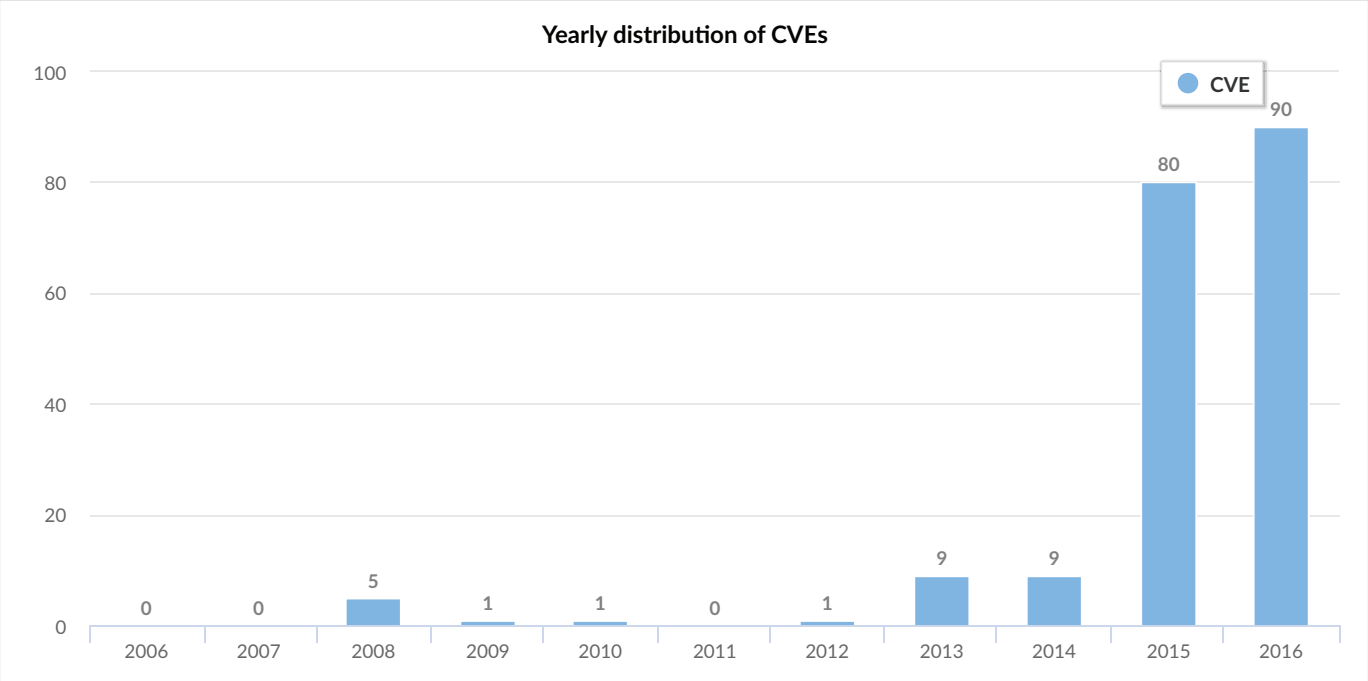


Fig. 13: Yearly distribution of CVEs. As the tested smartphone uses the Linux kernel version 3.18.14, which was released in May 2015, many of the vulnerabilities exposed in 2015 and 2016 are not patched yet.

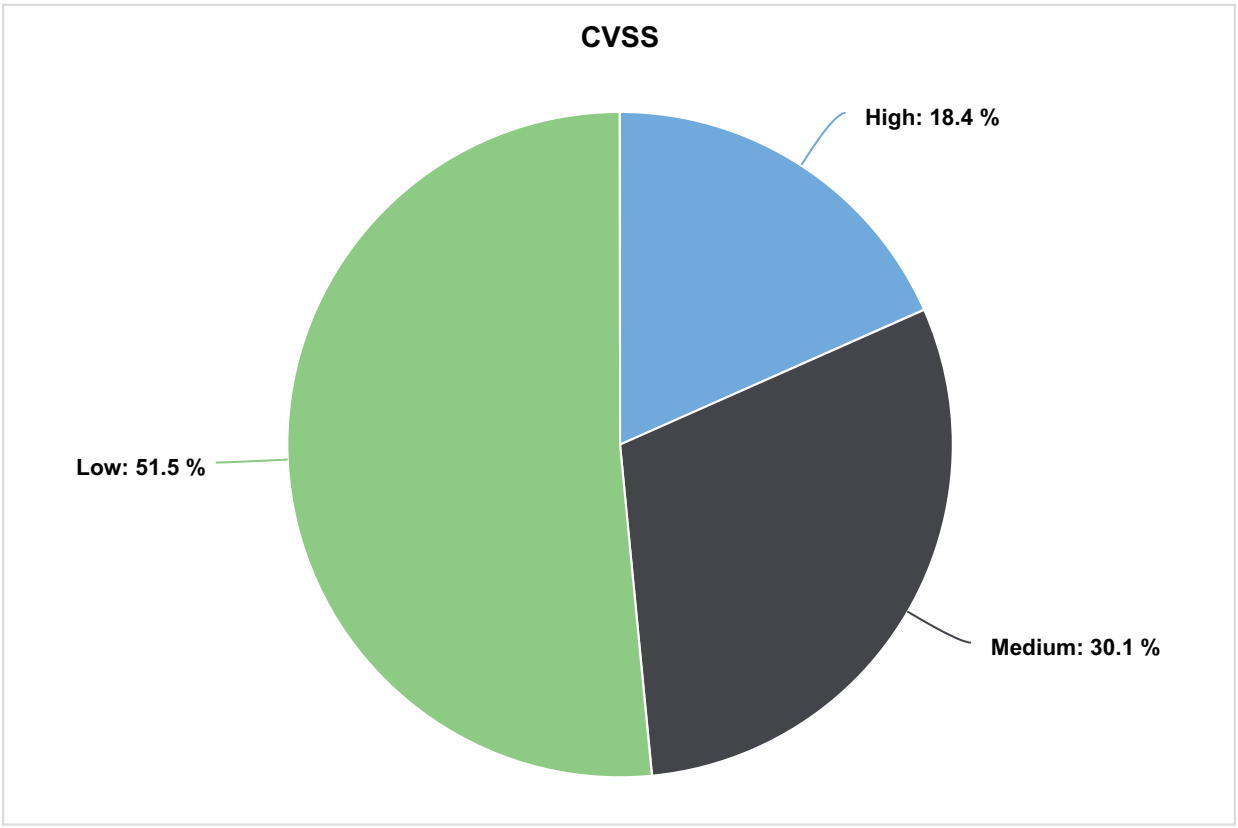


Fig. 14: The distribution of CVSS. Over 18 % of the detected CVEs are assigned with high severity score.

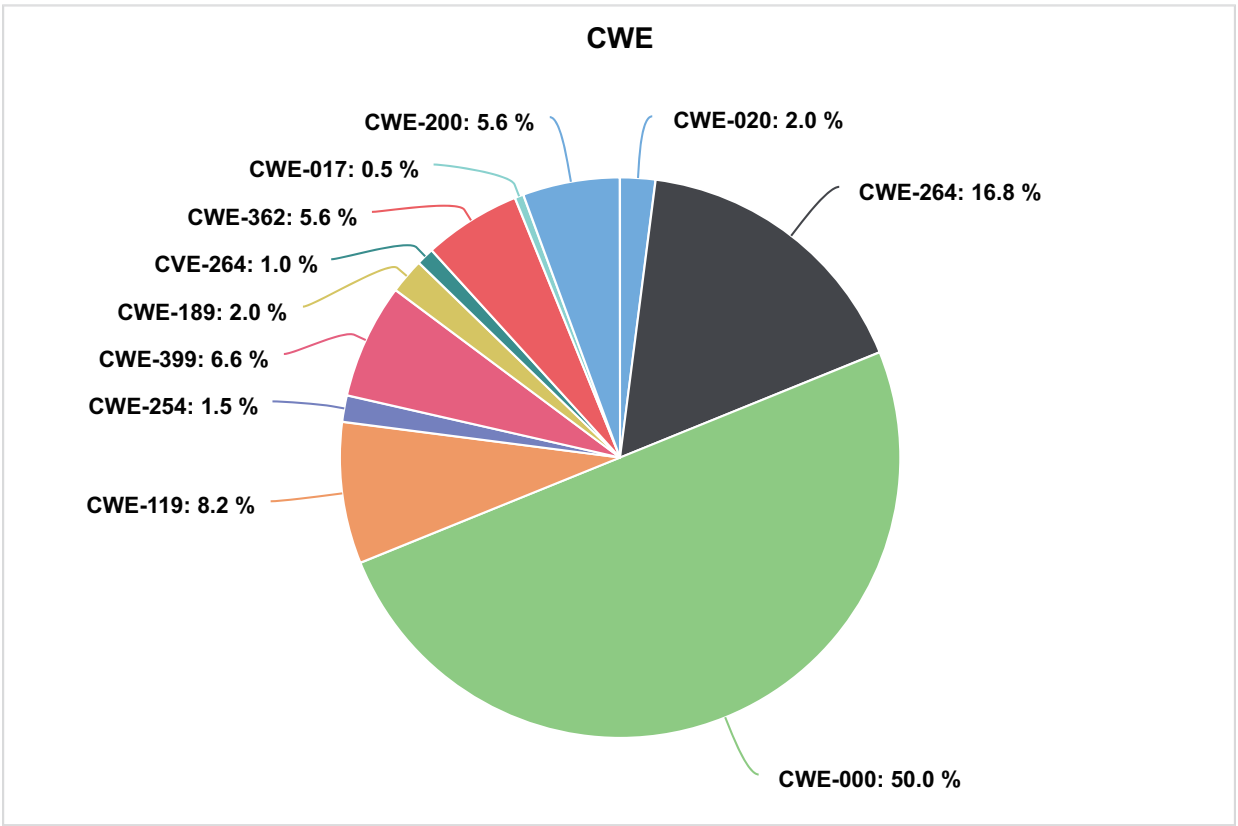


Fig. 15: The distribution of CWE. Permission-related vulnerabilities (CWE-264) are dominant in the Android smartphone.

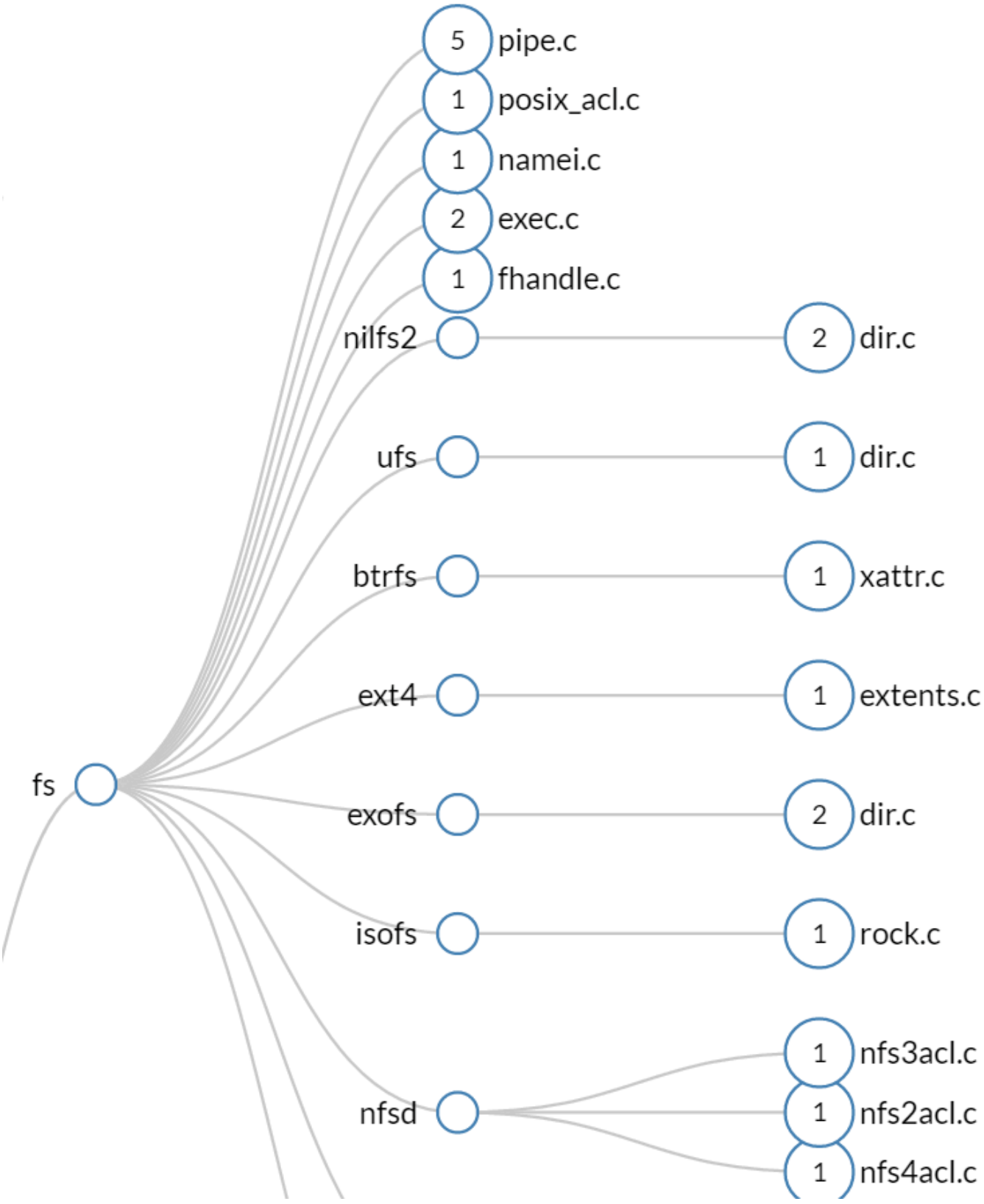


Fig. 16: A snippet of tree view. Internal nodes denote directories, and leaf nodes denote files. In this tree, dir.c file located under /fs/nilfs2/ has two vulnerable functions. Clicking the leaf nodes, users can also browse the vulnerable functions and corresponding patches.