IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

*Digital Object Identifier 10.1109/ACCESS.2017.DOI*

# xVDB: A High-Coverage Approach for Constructing a Vulnerability Database

**HYUNJI HONG, SEUNGHOON WOO, EUNJIN CHOI, JIHYUN CHOI AND HEEJO LEE**
Department of Computer Science and Engineering, Korea University, Seoul 02841, Korea
(e-mail: {hyunji_hong, seunghoonwoo, silver_jin, zzzmilky, heejo}@korea.ac.kr)

Corresponding author: Heejo Lee (e-mail: heejo@korea.ac.kr).

This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2019-0-01697 Development of Automated Vulnerability Discovery Technologies for Blockchain Platform Security, No.2022-0-00277 Development of SBOM Technologies for Securing Software Supply Chains, No.2022-0-01198 Convergence Security Core Talent Training Business, and No.IITP-2022-2020-0-01819 ICT Creative Consilience program).

**ABSTRACT** Security patches play an important role in detecting and fixing one-day vulnerabilities. However, collecting abundant security patches from diverse data sources is not a simple task. This is because (1) each data source provides vulnerability information in a different way and (2) many security patches cannot be directly collected from Common Vulnerabilities and Exposures (CVE) information (*e.g.*, National Vulnerability Database (NVD) references). In this paper, we propose a high-coverage approach that collects known security patches by tracking multiple data sources. Specifically, we considered the following three data sources: repositories (*e.g.*, GitHub), issue trackers (*e.g.*, Bugzilla), and Q&A sites (*e.g.*, Stack Overflow). From the data sources, we gather even security patches that cannot be collected by considering only CVE information (*i.e.*, previously untracked security patches). In our experiments, we collected 12,432 CVE patches from repositories and issue trackers, and 12,458 insecure posts from Q&A sites. We could collect at least four times more CVE patches than those collected in existing approaches, which demonstrates the efficacy of our approach. The collected security patches serves as a database on a public website (*i.e.*, IoTcube) to proceed with the detection of vulnerable code clones.

**INDEX TERMS** Open Source Software, Software security, Vulnerability database

## I. INTRODUCTION

As the open-source supply chain has been accelerated, the open-source vulnerabilities are also in the limelight. According to the "State of the Software Supply Chain Report" of Sonatype [20], 29% of popular open-source software (OSS) projects contain at least one known vulnerability that may become the attack surface to exploit the entire system (*i.e.*, called one-day exploits). In practice, cyberattacks which aimed at OSS projects have exponentially grown. For example, supply chain attacks, which are caused by the improper management of OSS, increased 650% in 2021 [20].

To prevent such attacks, developers mainly use the following two methods: (1) OSS version updates (*e.g.*, [5], [26]) and (2) fixing vulnerable source codes by leveraging Common Vulnerabilities and Exposures (CVE) information (*e.g.*, [9], [10], [27]). To proceed with such methods, especially the latter one, it is necessary to construct a large-scale vulnerability database, which is containing security patches and vulnerable codes [21], [23].

**Limitations of existing approaches.** Despite the importance of constructing a vulnerability database, existing approaches have limited in terms of collecting abundant security patches. Several approaches (*e.g.*, [11], [18], [21]–[23], [25]) attempted to construct vulnerability database, but they exhibited low patch collection coverage owing to the following two reasons: (1) limited data sources and (2) shallow scanning problems. Although vulnerability information is distributed among various data sources, previous approaches only focused on GitHub as their target data source for collecting security patches. In addition, most existing approaches considered only the security patches where the patch URL was provided in the NVD references, resulting in missing many security patches. Although some approaches [10], [25] took into account the hidden security patches that could be collected by searching for CVE ID keywords (in commit messages), simply matching keywords may lead to many false positives. Thus, we need to collect security patches that are not directly provided by considering various data sources.

**Our approach.** In this paper, we propose a high-coverage approach that collects known security patches by tracking multiple data sources. We construct a large-scale patch database called xVDB (**Ex**tended **V**ulnerability **D**ata**B**ase) by applying our approach. The key idea of our approach, which is distinguishable from existing approaches, is to collect known security patches by leveraging hidden connectivity between public vulnerability databases and security patches.

To address the limited data source problem, we consider the following three data sources: (1) *repositories*, *issue trackers*, and *Q&A sites*. Repositories (*e.g.*, GitHub) are most widely used when managing software source codes, and issue trackers (*e.g.*, Bugzilla) are widely leveraged for managing security bugs and vulnerabilities. Moreover, Q&A sites (*e.g.*, Stack Overflow) are not managed by publicly disclosed vulnerability database, such as the NVD, but insecure code snippets are being produced from such sites and propagated to OSS [6], [7]. Therefore, we determined that a wealth of security patches could be gathered from the aforementioned three data sources.

To address the shallow scanning problem, we first classify vulnerabilities based on the method of providing security patches: vulnerabilities with (1) direct patch links, (2) indirect patch links, and (3) invisible patch links. We then devise three patch collection methods according to each link type: (1) a method that directly collects security patches from CVE information (*i.e.*, direct patch links), (2) a method that collects security patches using hints (*e.g.*, commit ID) on the website provided by CVE information (*i.e.*, indirect patch links), and (3) a method that collects security patches from a data source by checking whether patches have security-related features, *e.g.*, CVE ID keywords and security sensitive APIs (*i.e.*, invisible patch links).

**Evaluation and findings.** In our experiments, we collected 12,432 CVE patches from repositories and issue trackers for the C, C++, Java, JavaScript, Python and Go languages. In addition, we collected 12,458 insecure posts from Q&A sites for C, C++ and Android posts. The collected security patches are at least four times more than those collected in the existing approaches, which is demonstrating the high patch collection coverage of our approach (see Table 2).

Our further analysis affirmed that xVDB exhibits the following five characteristics (see Section IV-C): (1) many security patches that urgently need to be patched (*i.e.*, medium and high severity) were collected in xVDB, (2) various types of vulnerabilities, including "Buffer Overflow" and "Out-of-bounds Read and Write", were gathered in xVDB, (3) more than half of the vulnerabilities were collected via indirect and invisible patch links, (4) most security patches were related to C/C++ languages, and (5) the reference site where the most patches were collected was GitHub, but a considerable number of security patches were collected via issue trackers.

We serviced xVDB as a database to detect vulnerable code clones on a public website (*i.e.*, IoTcube [8]) to contribute to the security of the software ecosystem.

This paper makes the following three contributions:

- We propose xVDB, a vulnerability database that is constructed using a high-coverage patch collection approach. The key idea of our approach is collecting known security patches by identifying hidden connectivity (*e.g.*, commit ID and CVE ID keywords) between public vulnerability databases and security patches.
- When we applied our approach, we collected 12,432 CVE patches from the repositories, issue trackers, and 12,458 insecure posts from the Q&A sites. Our approach could collect at least four times more CVE patches than those collected in existing approaches.
- We utilized xVDB as a database to detect vulnerable code clones on IoTcube [8], a public web platform for discovering security vulnerabilities in software.

## II. MOTIVATION

In this section, we introduce several terms used throughout this paper and then clarify our target problems.

### A. BACKGROUND AND TERMINOLOGY

#### 1) PUBLIC VULNERABILITY DATABASE

To mitigate risks caused by known security vulnerabilities, previously discovered vulnerabilities once discovered are managed through a public vulnerability database (*e.g.*, NVD [15], CVE MITRE [14], and CVE Details [4]) in the form of CVE.

Although there are slight differences between public vulnerability databases, the following pieces of information are commonly provided by them:

- **CVE ID:** A vulnerability unique identifier assigned by the MITRE corporation
- **Descriptions:** A summary of the overall introduction of each vulnerability, including affected products and attack vectors
- **Severity:** An indicator that represents the severity of a vulnerability (*e.g.*, CVSS)
- **Types:** A value indicating the type of vulnerability, such as buffer overflow or remote code execution. CWE is mainly used.
- **Affected software configurations:** The name and version information of the software that are affected by the vulnerability (*e.g.*, Common Platform Enumeration, CPE for short)
- **References:** A set of reference links related to the vulnerability. This includes URLs that contain the patch or a reproduction method for the vulnerability.

#### 2) KNOWN AND UNKNOWN VULNERABILITIES

We define a *known vulnerability* as a vulnerability that is managed by public vulnerability databases by assigning a CVE ID. All known vulnerabilities have corresponding security patches that are often disclosed through GitHub commits [11] or issue trackers such as Bugzilla [25]. By contrast,

we define an *unknown vulnerability* as a vulnerability that is not managed by CVE. Here we define that unknown vulnerabilities are patched secretly but are not managed by CVE from the public vulnerability database. This concept is somewhat different from a zero-day security vulnerability, which may still exist in the latest version of certain software without being patched.

#### 3) SECURITY PATCH

We define a security patch as a source code-level patch that is applied to resolve security issues [10], [27]. In general, security patches are provided in the form of "`diff`" of a code before and after applying the patch. For example, Listing 1 shows the security patch snippet for CVE-2021-41216, a heap buffer overflow vulnerability in TensorFlow.

**Listing 1: Security patch snippet for CVE-2021-41216.**

```
1  diff --git a/tensorflow/core/ops/array_ops.cc
2             b/tensorflow/core/ops/array_ops.cc
3  index 64bd4f3847854..14c9efae1ddd3 100644
4  --- a/tensorflow/core/ops/array_ops.cc
5  +++ b/tensorflow/core/ops/array_ops.cc
6
7  @@ -168,7 +168,7 @@ Status TransposeShapeFn(...) {
8
9     for (int32_t i = 0; i < rank; ++i) {
10       int64_t in_idx = data[i];
11 -   if (in_idx >= rank) {
12 +   if (in_idx >= rank || in_idx <= -rank) {
13       return errors::InvalidArgument("perm dim ",
               in_idx, " is out of range of input rank ",
               rank);
14 }
```

From the security patch, we can obtain several pieces of information for efficient vulnerability management. For example, we can identify the vulnerable and patched source files (*i.e.*, `array_ops.cc`), the index values of files before and after applying the patch (*i.e.*, line #3 in Listing 1), the code line numbers to which the patch was applied (*i.e.*, 7 lines from line #168 in the "`array_ops.cc`" file), and the actual code lines that were added or deleted in the security patch (*i.e.*, lines #11 and #12 in Listing 1).

### B. GOAL AND CHALLENGE STATEMENT

**Goal.** In this paper, we construct a vulnerability database (called xVDB) by collecting information on known vulnerabilities. Specifically, our main goal is to collect security patches of vulnerabilities at the source code levels, which can be used to detect one-day vulnerabilities [9], [10], [27]. Subsequently, xVDB can assist in detecting propagated vulnerabilities (*i.e.*, 1-day vulnerabilities) and consequently, can be used to mitigate threats caused by vulnerable code reuse.

Constructing a rich and well-refined vulnerability dataset is important because it has a significant impact on the vulnerability detection process. Since most of the existing one-day vulnerability discovery techniques (*e.g.*, [10], [27]) detect vulnerabilities based on the collected vulnerability data, failing to construct an abundant and well-refined vulnerability dataset may compromise the vulnerability detection accuracy (*e.g.*, missing many vulnerabilities).

**Challenges.** The collection of security patches is not a simple task. The biggest obstacle is the diversity of data sources. Vulnerability information is distributed among various sources, such as repositories (*e.g.*, Git), vulnerability databases (*e.g.*, NVD), and issue trackers (*e.g.*, Bugzilla), in various forms.

Since the data sources provide vulnerability information in different ways and are complementary to one another, considering only one data source may result in a biased and insufficient dataset. For example, Kim *et al.* [10] collected vulnerability information only from GitHub, and Woo *et al.* [25] considered GitHub and issue trackers. However, a recent study (*i.e.*, PATCHSCOUT [21]) demonstrated that approaches that considered only a part of data sources showed a low patch collection coverage (*i.e.*, at most 53%) and lack of accuracy (*i.e.*, only 47% of the collected data was accurate). In particular, in PATCHSCOUT, a significant amount of human intervention is required to collect security patches.

Moreover, each data source may not explicitly provide vulnerability information. For example, a commit that patched a specific vulnerability exists on GitHub, and the vulnerability is registered in a public vulnerability database with a specific CVE ID. However, there may not be a direct connection between the patch commit URL and corresponding CVE vulnerability (*i.e.*, invisible links). Therefore, there is a need for an automated method that can collect security patches while considering various data sources.

## III. METHODOLOGY

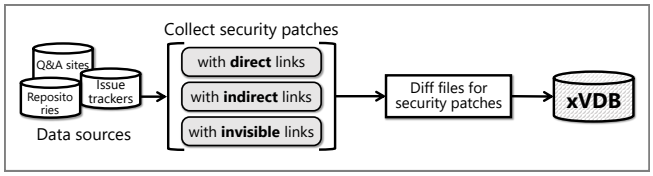


**FIGURE 1:** High-level workflow of xVDB construction.

In this section, we introduce the methodology for constructing xVDB. The high-level workflow for constructing xVDB is shown in Figure 1. The main goal of our approach is to collect known security patches by tracking multiple data sources. To address the low coverage from limited data sources, we collect security patches from three target data sources: *repositories*, *issue trackers*, and *Q&A sites*. Then, we collect known security patches by identifying hidden connectivity between public vulnerability databases and security patches. We describe the collection method for obtaining security patches directly and indirectly from CVE information (see Section III-B2, and Section III-B3). In addition, we introduce the method of collecting security patches that are invisible in CVE information but can be collected from data sources (see Section III-B4).

### A. DEFINITION

This section introduces the key definitions used in our approach.

**Definition 1) Basic terms.** We first define a few terms upfront. *CVE info page* denotes an information page for each CVE in the public vulnerability database (*e.g.*, NVD and CVE MITRE). *CVE patch page* refers to the page with the patch of the CVE vulnerability. *Self-managed repository* refers to a repository where the software manages its source code through its own management system rather than being managed by a major hosting platform such as GitHub.

**Definition 2) Classification of patch links.** We categorize security patches based on the method of providing security patches from the CVE info page. Since each target data source has the characteristic of providing security patches, and therefore we propose the appropriate collection methods by categorizing patches. To this end, we classified vulnerabilities into the following three types: vulnerabilities with (1) direct patch links, (2) indirect patch links, and (3) invisible patch links (see Figure 2).

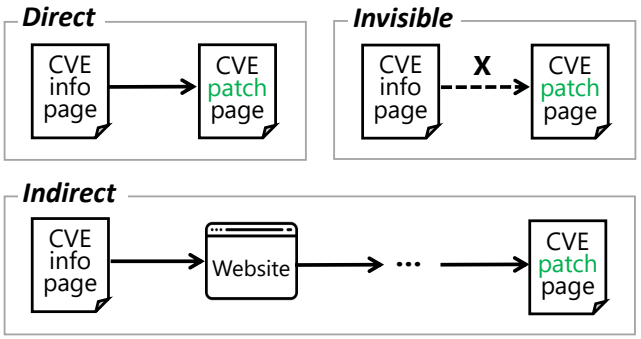


**FIGURE 2:** Vulnerability classification according to the way that the security patch is served.

**(1) Direct patch link**: This refers to the case where the CVE patch page is directly provided by the CVE info page.

**(2) Indirect patch link**: This refers to the case whre the CVE patch page is indirectly provided by the CVE info page, *e.g.*, the CVE info page provides a website containing the security patch URL such as Bugzilla bulletin board.

**(3) Invisible patch link**: This refers to the case where the link between the CVE patch page and the CVE info page is invisible.

**Definition 3) Target data sources.** To collect security patches, we target the following three data sources: (1) repositories, (2) issue trackers, and (3) Q&A sites. Known vulnerabilities and their corresponding patches are primarily shared via the three target data sources. Here, we explain in detail the reason for selecting each data source.

**(1) Repositories** (*i.e.*, software repositories) are storage locations for the collection of files of various different versions of software programs. Since many software products are developed in collaboration with developers, repositories are most widely used when managing software programs. In particular, when a vulnerability is discovered in the software source code, developers tend to provide a patch code with a relevant message via the repositories. Therefore, we first select the repositories as our target data source. To create and manage the repositories, version control systems such as Git, Subversion (*i.e.*, SVN) and Mercurial are used. In particular, we focus on Git as our target data source.

**(2) Issue trackers** are tools for tracking bugs and managing other issues in software vendors. Each vendor tends to manage issues with their own issue tracking systems (*e.g.*, Mozilla manages with *bugzilla.mozilla.org*). Since a considerable number of vulnerabilities were issued and reported to CVE (which accounts for 5% of the total CVEs), we select *issue tracker* as our target data source.

**(3) Q&A sites** (*e.g.*, Stack Overflow) are platforms that discuss code problems, and many developers rely on such platforms [17]. Although insecure code snippets from these platforms may be conveyed to OSS [6], [7], the insecure code snippets are not managed by the public vulnerability database. Therefore, we also consider *Q&A sites* as our targe data source. In particular, we focus on Stack Overflow as our target data source.

### B. COLLECTING SECURITY PATCHES

In this section, we introduce our key idea that was extracted from our observations, and then propose patch collection methods for each link type.

#### 1) KEY IDEA

Observation

**[Direct patch link]**
The CVE info page provides the CVE patch page via references (*e.g.*, NVD reference URL), and the URLs have certain patterns.

**[Indirect patch link]**
To provide security patches, the CVE info page provides websites (*e.g.*, issue trackers and self-managed repositories) via references. The websites may contain a hint introducing a CVE patch page (*e.g.*, Bug ID and commit).

**[Invisible patch link]**
Even if there is no link that introduces the CVE patch page in a CVE info page, the CVE patch page may contain the corresponding CVE ID.

We first observed that a *direct patch link* is provided in certain patterns, depending on the hosting platform (*e.g.*, GitHub, GitLab, Cgit, and GitWeb). The patterns are that the patch links contain the git platform domain name (*e.g.*, github.com and gitlab.com) and the string "commit". For example, patch links hosted on GitHub are provided in the following format: *https://github.com/user/repo/commit/commit_ID*.


**FIGURE 3:** Example of where patches can be collected through the website provided by the CVE info page.

In addition, we observed that some websites provided by the CVE info page contain a hint that introduces the CVE patch page. For example, when an issue is uploaded to an issue tracker site, developers tend to leave the comments with patch commits on the issue page. Figure 3 presents an example of an *indirect patch link*. The CVE info page provides the issue link as references, and the issue post contains corresponding patch links raised by comments.

Lastly, we observed cases in which a CVE ID was found on the CVE patch page, even though there is no link related to the security patch on the CVE info page. This is because when known vulnerabilities are patched, developers tend to leave a commit message with their CVE ID.

**Listing 2: OpenVPN commit #cb4e35e.**

```
[Commit message]

Fix potential double-free in --x509-alt-username
(CVE-2017-7521)

[NVD Descriptions of CVE-2017-7521]

OpenVPN versions before 2.4.3 and before 2.3.17
are vulnerable to remote denial-of-service due to
memory exhaustion caused by memory leaks and
double-free issue in extract_x509_extension().
```

As an example of an *invisible patch link*, we introduce the case of CVE-2017-7521, a double-free vulnerability (see Listing 2). Although the CVE-2017-7521 info page does not provide a link for security patches, we can discover the security patch by searching with CVE ID.

Therefore, we collect security patches by leveraging these observations. Figure 4 illustrates our model of collecting patches using a hidden connectivity between the CVE info page and the CVE patch page. For indirect patch links, some websites provide hints (*e.g.*, commit ID) leading to CVE patches; this means that there is a hidden connectivity between those websites and CVE patches. Accordingly, we collect corresponding CVE patches by scanning data sources

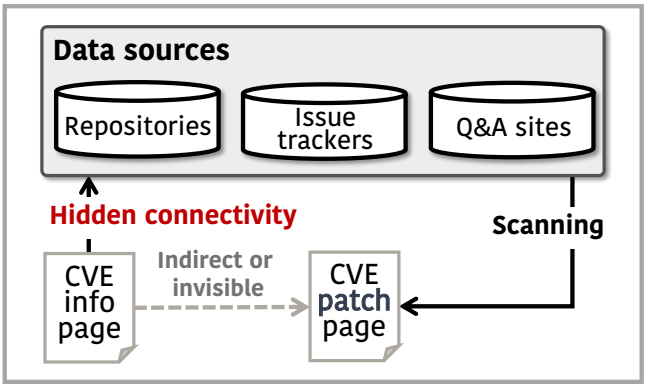


**FIGURE 4:** Patch collection model using a hidden connectivity between the CVE info page and the CVE patch page.

with the obtained hints. For invisible patch links, we consider the CVE ID keyword as a hidden connectivity between the CVE patch and the CVE info page. Therefore, to collect corresponding CVE patches, we utilize the CVE ID keyword to scan the data sources.

However, not all data sources can apply all link types to collect security patches. Because only six CVE vulnerabilities were referenced in the case of Q&A sites, it is difficult to apply the collection method with a *direct or indirect patch link*. In addition, it is rare to provide patches directly to the issue trackers and therefore we could not apply the collection method with a *direct patch link* in the issue tracker cases. Therefore, we summarize the applicable data sources by the link type in Table 1.

**TABLE 1:** Applicable data sources by the link type.

| Data source | Type of link | | |
|---|---|---|---|
| | Direct | Indirect | Invisible |
| Repositories | ✔ | ✔ | ✔ |
| Issue trackers | | ✔ | ✔ |
| Q&A sites | | | ✔ |

### 2) DIRECT PATCH LINK COLLECTION

We first introduce a method for collecting security patches with direct patch link. Given a CVE info page as an input, we collect the CVE patch page by checking whether it is a direct patch link hosted on Git platforms (*e.g.*, Github, Gitlab, Cgit and Gitweb). Here are the steps for collecting patches:

(1) *Search the URL in reference links:* We first collect commit URLs with the keyword "commit" and the name of Git platforms. For example, the CVE-2020-14147 info page provides the CVE patch page of GitHub with a direct link with the URL "*https://github.com/redis/redis/commit/ef764dd*".

(2) *Download the repository in a local environment:* This step is a prerequisite for extracting security patches. To download the repository into the local environment, the command *git clone repository_url* is required.

(3) *Extract the diffs:* We then extract the diffs related to the retrieved commits using the command *git show commit_id* on a specific cloned repository. This command shows the security patch codes in a unified diff format and commit messages.

While security patches can be extracted by crawling the URL retrieved in the first step, some platforms such as GitWeb require preprocessing to extract the patches (*e.g.*, replacing with a URL that can be crawled in the form of plain text). Therefore, we propose the steps to extract a security patch using Git commands.

### 3) **INDIRECT** PATCH LINKS COLLECTION

To collect security patches with indirect patch link, we propose an *in-depth scanning method* that tracks the CVE patch page by analyzing the website URL, given the CVE info page. As explained in Section III-B1, issue trackers (*e.g.*, Bugzilla and GitHub issue page) and self-managed repositories are considered in this phase. Given a CVE info page as an input, the security patches are collected as follows:

(1) *Crawl website URLs provided by the CVE info page:* To analyze the website URL, we first crawl the HTML contents of the URL. We crawl the URL by requesting it using a simple crawler (*e.g.*, `BeautifulSoup`).

(2) *Extract information addressing the CVE patch page:* Next, we find the information that may link to the CVE patch page. There are two cases of collecting information:

(2-1) Case with a direct link (*e.g.*, Git commit URL) of the CVE patch page (see Figure 3). If the direct patch link is identified in the website contents, we can apply the same method for the collection with direct patch links (see Section III-B2).

(2-2) Case as a hint to the CVE patch page. We collect information that can be used to find patch commits on the website. For example, Mozilla records bug IDs in the corresponding patch commit messages. Therefore, we can leverage the bug IDs to find security patches.

(3) *Search the patch commits and extract diffs:* We then search the patch commits with the information extracted from the previous steps. For example, if a bug ID is detected through Step (2-2), we can extract the corresponding security patches by executing the command *git log –grep='bug ID'*.

Since we do not know what information is leveraged to find the security patch commits in Step (2-2), we conducted a preliminary investigation of the information to be extracted.

As an example, we introduce CVE-2020-11655, a vulnerability that provides a patch to the SQLite software (see Figure 5). Although the source code of SQLite is hosted on GitHub, vulnerabilities of SQLite are managed by its own management system. By leveraging the ID (*i.e.*, SHA3-256) that is provided in the website URL, we searched for relevant commits that contain such ID in the commit messages. Finally, we can extract the diffs using the retrieved commit.


**FIGURE 5:** Example of a case where security patches can be obtained externally with information in the websites provided by the CVE info page.

### 4) **INVISIBLE** PATCH LINKS COLLECTION

To collect patches with invisible links, we leveraged our previous work, DICOS [7], which is an accurate approach for discovering insecure code snippets in Stack Overflow posts.

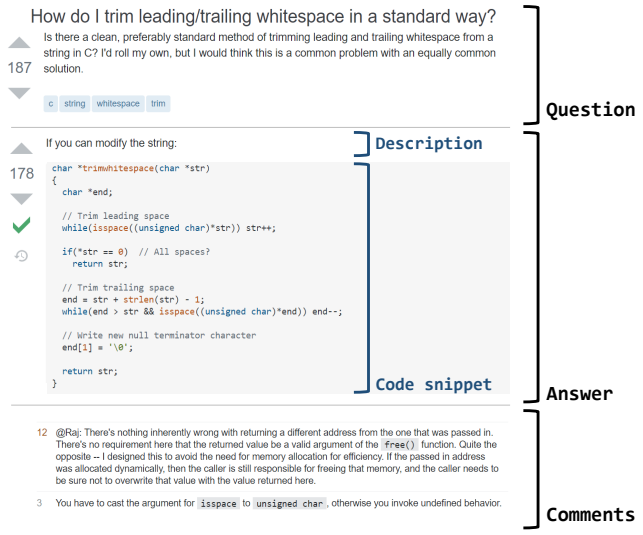


**FIGURE 6:** Example Stack Overflow post (#122721). We divide a post into three parts: question, answer, and comments; the answer is further subdivided into code snippet and description (*i.e.*, narrative part excluding code snippets).

By leveraging DICOS, we can collect security patches with missing links in the Q&A sites. The key idea of DICOS for discovering insecure code snippets is leveraging user discussions in Stack Overflow. In general, an answerer edits their code snippets when they notice that their code has a flaw, such as a security issue and thereafter, they leave all the edit

logs in their post. Inspired by this process, DICOS analyzes the change history of the post, as it provides significant hints for discovering insecure code snippets. In a nutshell, DICOS first extracts the the change history (*i.e.*, diffs between the oldest and latest revisions) from the Stack Overflow post for the description, code snippets, and comments (see Figure 6). DICOS then discovers insecure code snippets by analyzing whether the extracted diffs are intended to fix a security issue based on the selected features (*i.e.*, security-sensitive APIs, security-related keywords, and control flow information).

In addition, we collect security patches with invisible patch links in the repositories and issue trackers by leveraging DICOS. Here, we consider a commit as the same concept as the change history of DICOS and therefore analyze the commit message and diff of the source codes. Since the CVE ID can be a hint for finding relevant commits, we search patch commits by analyzing whether the commit message contains "CVE-20" (the command *git log –grep='CVE-20'* is used). The algorithm for the patch collection methods is given in Algorithm 1.

**Algorithm 1:** Algorithm for collecting security patches.

```
Input: V, C, R
// V: Vulnerability, C: CVE info page,
// R: Repository reporting V
Output: P
// P: Security patch for V
procedure ExtractingPatch(V, C, R)
    Ref ← References(V, C)
    for URL in Ref do
        if ("git" in URL) and ("commit" in URL) then
            // Collect P with direct patch links
            P ← Crawl(URL)
        else
            // Collect P with indirect patch links
            if GitURL in Visit(URL) then
                P ← Crawl(GitURL)
            else if H in Visit(URL) then
                // H: Hints for detecting patches
                (e.g., Commit ID or Bug ID)
                for C_m in R do
                    // C_m: Commit
                    P ← GetPatchCommit(C_m, H)
    // Collect P with invisible patch links
    for C_m in R do
        if "CVE-20" in C_m then
            if (IsControlFlowChanged(C_m) or
              IsSecurityAPIChanged(C_m)) then
                P ← C_m
    return P
```

## IV. FINDINGS

In this section, we provide the analysis results related to the following four questions:

- **Q1.** How do we implement our approach for constructing xVDB? (Section IV-A)
- **Q2.** How many security patches have we collected? (Section IV-B)
- **Q3.** What are the characteristics of the collected security vulnerabilities? (Section IV-C)
- **Q4.** How is our approach applied in the real world? (Section IV-D)

### A. IMPLEMENTATION OF OUR APPROACH

Based on the methodology presented in Section III, we constructed xVDB collected from the repositories, issue trackers, and Q&A sites. For repositories and issue trackers, we targeted the C, C++, Java, JavaScript, Python, and Go languages because our additional experiment affirmed that these six languages belong to the top languages of the patches reported as CVE. For Q&A sites, especially Stack Overflow, we targeted C, C++, and Android posts because the reuse of small pieces of code is prevalent in the software [6], [10], [25], [26]. Note that the design of our approach can be applied to any programming language.

Our approach was implemented on approximately 1,000 lines of Python code, excluding the external libraries (*e.g.*, `BeautifulSoup`). We used DICOS, an open-source tool, to collect security patches with invisible patch links on the Q&A sites. The source code for DICOS is available at https://github.com/hyunji-hong/DICOS-public.

### B. COVERAGE OF xVDB

In our experiments, we collected **12,432 CVE patches** from the repositories and issue trackers, and **12,458 insecure posts** from the Q&A sites. To demonstrate that our approach has higher coverage than those proposed in previous approaches, we compared the security patches in xVDB with those of existing approaches.

**Methodology for comparison.** We reviewed several approaches that attempted to collect security patches [10]–[12], [18], [21], [23], [25]. We classified the number of security patches collected by existing approaches according to the link type we defined. Note that we did not consider the unknown security patch discovery and therefore, we performed comparison experiments for the known security patch discovery. Table 2 shows the results of the experiments.

**Result analysis.** We confirmed that our approach significantly outperformed existing approaches, as we could collect at least four times more than the other approaches and even collected insecure posts from the Q&A sites.

We first confirmed that CVE patches with a direct patch link were collected more than the other approaches. Although we considered various programming languages, most patches were for C and C++ languages (accounting for 81% of the total; details are presented in Section IV-C4). Considering that most of the existing approaches only collected C/C++ security patches, it can be seen that our approach still collected

**TABLE 2:** Coverage of xVDB.

| Approach | Direct | Indirect | | Invisible | | | Total |
|---|---|---|---|---|---|---|---|
| | R* | R* | IT† | R* | IT† | QA‡ | |
| [18] | 718 | X | X | X | X | X | 718 CVE |
| [11] | 3,094 | X | X | X | X | X | 3,094 CVE |
| [10] | X | X | X | 3,551 | X | X | 3,551 CVE |
| [12] | 809 | X | X | X | X | X | 809 CVE |
| [25] | 3,246 | X | X | X | 2,425 | X | 5,671 CVE |
| [23] | 4,076 | X | X | X | X | X | 4,076 CVE |
| **xVDB** | 6,387 | 1,644 | 3,020 | 2,966 | 1,766 | 12,458 | **12,432 CVE 12,458 Posts** |

R*: Repositories
IT†: Issue trackers
QA‡: Q&A sites

more patches; this is because, existing approaches mainly considered GitHub among the various Git platforms.

Although most approaches did not cover the collection methods proposed in our approach (*e.g.*, collecting indirect and invisible patch links), VUDDY [10] and V0Finder [25] collected patches with invisible patch links. We confirmed that VUDDY and V0Finder collected more security patches (*i.e.*, 3,551 and 2,425 CVE patches, respectively) than xVDB. This is because they collected security patches by searching for the keyword "CVE-20" in commit messages, which may easily produce false positives. However, our approach reduces such false positives by analyzing whether patches contain "CVE-20" as well as control flow changes or security-sensitive API changes (see Section III-B4).

Figure 7 represents the comparison results of xVDB and the existing approach [25] by each link type. Here, some patches coexist on multiple link types, thus we consider the following priority: (1) direct, (2) indirect, and (3) invisible. As a result, 6,387 CVE patches (51%) are categorized as the direct link, 2,358 CVE patches (19%) belong to the indirect link, and the remaining 3,687 CVE patches (30%) are classified as the invisible patch link. We selected V0Finder [25], which collected the most security patches (*i.e.*, 5,671 security patches) among existing approaches, as a comparison target.

From our experiments, our approach improved 1.9 times and 1.5 times of the direct link and invisible links stored in V0Finder, respectively; Our approaches collected 2.1 times more security patches than V0Finder.

As shown in Figure 7 and Table 2, most existing approaches could cover only some parts of direct and invisible patch links, resulting in collecting fewer patches than we stored in xVDB; this result demonstrates that our approach has higher coverage compared to the existing approaches.

### C. CHARACTERISTICS OF PATCHES ON *xVDB*

To identify the characteristics of security patches in xVDB, we examined security patches from six perspectives. To state the conclusion first, we introduce the key findings:

(1) Many security patches that urgently need to be patched were collected in xVDB. (Section IV-C1)

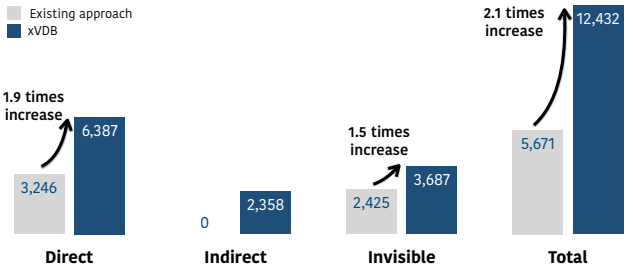


**FIGURE 7:** Comparison results of xVDB and the existing approaches [25] by link type.

(2) Many security patches in xVDB were classified as types that require boundary checking. (Section IV-C2)

(3) More than half of the security patches were collected via indirect and invisible patch links. (Section IV-C3)

(4) Most security patches (81%) were related to C/C++ languages. (Section IV-C4)

(5) The reference site where the most patches were collected was GitHub, but a considerable number of security patches were collected via issue trackers. (Section IV-C5 and Section IV-C6)

#### 1) SEVERITY OF CVE PATCHES

To identify how critical the CVE vulnerabilities collected in xVDB are, we used the Common Vulnerability Scoring System (CVSS), which is a standard vulnerability metric.

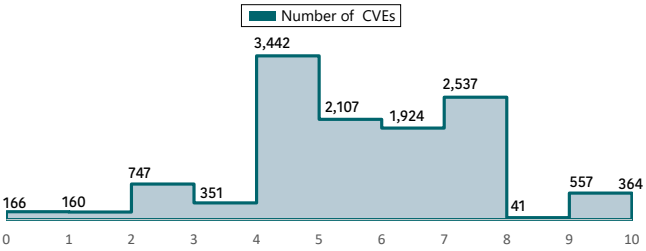


**FIGURE 8:** CVSS distribution of CVE patches in xVDB.

Figure 8 represents the distribution of CVSS scores in xVDB (specifically, we measured with CVSS version 2). Since the scores are represented to one decimal place, we aggregated the total number of CVE within the range (*e.g.*, score 7.8 is counted as score 7). As shown in Figure 8, we collected vulnerabilities with various levels of severity, rather than focusing on a specific level. Most vulnerabilities in xVDB are distributed in the medium level (*i.e.*, low: 0 to 3.9, medium: 4 to 6.9, and high: 7 to 10).

We also compared our results with the vulnerabilities disclosed by CVE MITRE [14] (see Figure 9); we confirmed that 6% to 7% vulnerabilities were collected for each severity level. This further represents that the reason most vulnerabilities in xVDB are distributed at the medium severity level is that a large portion of disclosed vulnerabilities also belong to the medium severity level. As 89% of the security patches (11,008) collected in xVDB showed medium or high

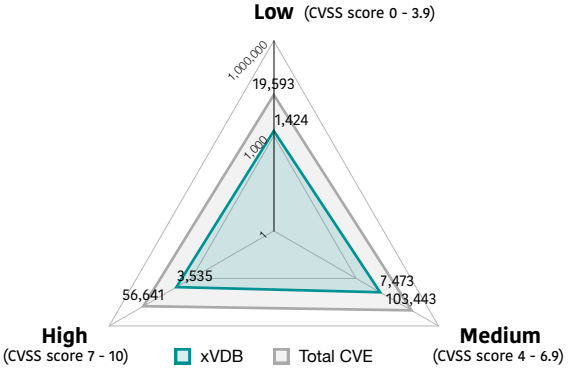


**FIGURE 9:** Comparison results of vulnerabilities in xVDB and vulnerabilities disclosed by CVE MITRE [14] based on CVSS (logarithmic scale).

severity, we determined many vulnerabilities that needed to be patched urgently in the real-world software ecosystem were collected in xVDB.

### 2) TYPES OF CVE PATCHES

To identify the type of each CVE patch, we used Common Weakness Enumeration (CWE) assigned to each CVE, which is a standard for software weakness types. With the collected CVE patches, we simply counted the number of CWE, and confirmed that 202 unique types of CWE were detected in xVDB. Table 3 represents the top 10 CWE types in xVDB.

**TABLE 3:** Top 10 CWE distribution discovered in xVDB.

| Rank | #CWE ID | #Weakness name | #Counting |
|---|---|---|---|
| **1** | CWE-119 | Buffer Overflow | 1,615 |
| **2** | CWE-787 | Out-of-bounds Write | 871 |
| **3** | CWE-125 | Out-of-bounds Read | 853 |
| **4** | CWE-20 | Improper Input Validation | 755 |
| **5** | CWE-200 | Information Disclosure | 586 |
| **6** | CWE-264 | Access Control Error | 571 |
| **7** | CWE-476 | NULL Pointer Dereference | 540 |
| **8** | CWE-79 | Cross-Site Scripting | 432 |
| **9** | CWE-416 | Use After Free | 423 |
| **10** | CWE-190 | Integer Overflow | 334 |

From the results, we confirmed that various types of vulnerabilities exist in xVDB. The most frequently appearing type is "Buffer Overflow", which can cause a system crash and therefore, requires additional boundary checking or avoidance of using standard library functions (*e.g.*, scanf and gets). Furthermore, we confirmed that "Out-of-bounds Write and Read" also account for a large part, and they also require additional boundary checking.

### 3) YEAR DISTRIBUTION OF CVE PATCHES

We then investigated the distribution of years in which security patches were disclosed from 1999 to 2022. The results are depicted in Figure 10.

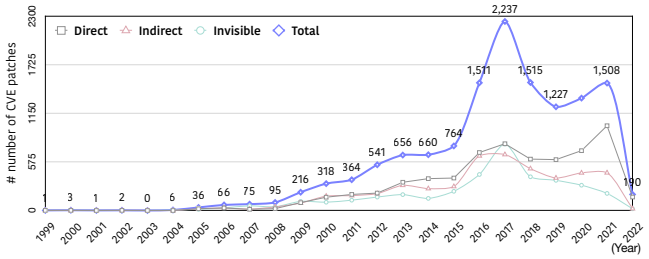


**FIGURE 10:** Year distribution of CVE patches in xVDB. CVE patches collected via direct, indirect, and invisible links and total CVE patches in xVDB are represented.

Although direct links are the most collected link type each year, indirect and invisible links also accounted for a large proportion of the total each year. As an example, in 2017, only 270 CVE patches could be collected when collecting patches via direct links. However, since we covered even indirect and invisible links, we could collect 2,237 CVE patches. This result affirmed that our approach could collect more security patches than the existing approaches that collected only via direct patch links.

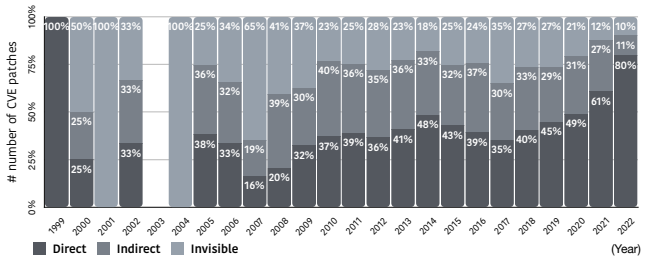


**FIGURE 11:** Cumulative graph of CVE patches collected by three link types by year (direct, indirect, and invisible patch links).

The cumulative graph by year of each link type is depicted in Figure 11. When examining only the ratio of link types, we can confirm that the proportion of direct patch links is gradually increasing. This indicates that as developers become more concerned with security, more CVE vulnerabilities are being reported and managed with direct patch links. Nevertheless, in most cases of each year, approximately half of the vulnerabilities could be collected via invisible or indirect patch links. Therefore, this suggests the need for an approach that even collects hidden security patches, such as the approach introduced in this paper.

When compared with the total number of reported CVEs disclosed via the CVE MITRE [14], we confirmed that our approach could collect an average of 6% of security patches for each year. For example, in 2021, we found 1,526 security patches (7.5%) among the 20,168 disclosed vulnerabilities. Although this ratio does not seem large, in fact, considering that the proportion of vulnerabilities that release patches is not very high, this is a sufficiently significant number. We discuss this in Section VI.

#### 4) LANGUAGE DISTRIBUTION OF CVE PATCHES

Next, we analyzed the language distribution of security patches in xVDB.

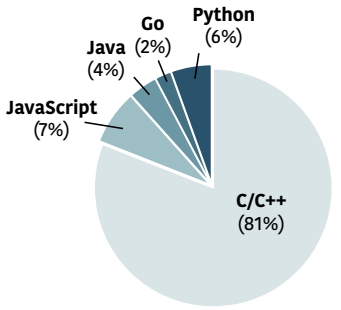


**FIGURE 12:** Language distribution of CVE patches in xVDB.

As shown in Figure 12, C and C++ accounts for a large portion (81%) of vulnerabilities in xVDB, *i.e.*, 10,067 out of 12,432 security patches. While many existing approaches focused only on the C and C++ languages for constructing the vulnerability database, we confirmed that other languages accounted for approximately 19% of the total CVE patches (*i.e.*, accounting for 2,365 CVE patches). Therefore, we need a technique that can collect security patches of various languages, and in that regard, we can claim the superiority of our approach, which is not limited by programming language.

#### 5) DATA SOURCE DISTRIBUTION OF CVE PATCHES

We further analyzed data sources for security patches in xVDB, to identify which data sources the patches were collected from.

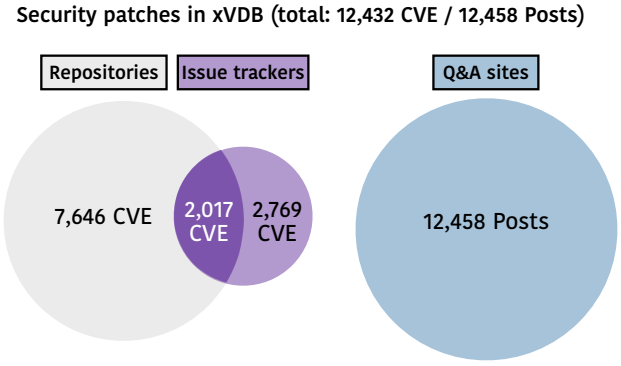


**FIGURE 13:** Illustration for the coverage of the data sources (repositories, issue trackers, and Q&A sites).

We confirmed that 12,432 CVE patches were collected from the repositories and issue trackers, and 12,458 posts were collected from the Q&A sites (see Figure 13). Most of the CVE patches were collected from repositories (9,663 patches, 78% of the total), and 4,786 patches were collected from issue trackers. We also confirmed that 42% (2,017) of the patches collected from the issue tracker were identically collected from the repositories. This is because a considerable amount of vulnerabilities are reported to the issue trackers before being reported to CVE MITRE [14] and thus patches are delivered to the repositories and issue trackers simultaneously.

As an example, the security patch of CVE-2020-35492 was collected from both the repository and the issue tracker. This vulnerability is related to Cairo software, an open source graphics library, and causes a stack buffer overflow. At first, a security issue was reported on the issue tracker [2] on November 2020. A corresponding patch was released to the Cairo repository [3] on December 2020, and thereafter the patch was provided as a direct link to the issue tracker. This issue was reported and CVE issued on March 2021 in CVE MITRE [14] with the issue tracker URL; and after a month, the CVE reference site was modified to contain the direct patch commit url as well.

However, it is worth noting that 58% (2,769) of CVE patches collected from issue trackers could not be collected in the repositories via direct and invisible patch links. This suggests that issue trackers should also be considered as target data sources, as we can collect a significant number of patches.

#### 6) REFERENCE SITE DISTRIBUTION OF CVE PATCHES

Finally, to identify which reference sites provide security patches, we investigated the reference site distribution of the CVE patches by analyzing 12,432 security patches in xVDB. Note that one CVE may contain multiple security patches. We collected the URLs from which CVE patches can be extracted and classified them into the eight categories. Figure 14 represents the measurement results.

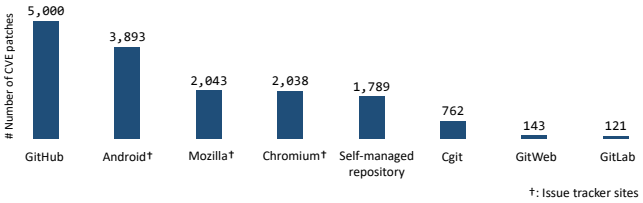


**FIGURE 14:** Distribution of reference sites of CVE patches in xVDB.

Not surprisingly, GitHub provided the highest number of security patches; this is because many OSS projects are hosted via GitHub, and thus security issues are mainly fixed with GitHub commit. Moreover, our experimental results show that issue trackers (*e.g.*, Android, Mozilla, and Chromium) and self-managed repositories (*e.g.*, SQLite) account for a large proportion. This indicates the need to collect patches from various data sources, which reveals the efficacy of our approach.

### D. APPLICATION

xVDB has been serviced online for free (at IoTcube [8]) since 2016. IoTcube provides several tools for detect-

ing vulnerabilities. Specifically, xVDB is leveraged as a database to detect vulnerable code clones, in conjunction with VUDDY [10]. Since 2016, over 20K users including commercial software developers, open-source committers, and IoT device manufacturers have tested our platform. Figure 15 shows the main page of the IoTcube platform.

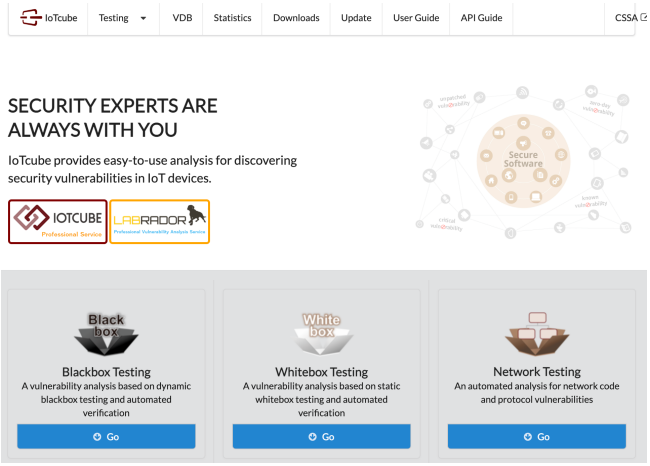


**FIGURE 15:** Main page of IoTcube.

IoTcube provides the statistics of xVDB; vulnerability statistics by language, vulnerability statistics by repository, and vulnerability statistics by year. Figure 16 illustrates the vulnerability statistics by language and year, and Figure 17 shows the vulnerability statistics by repository in xVDB.

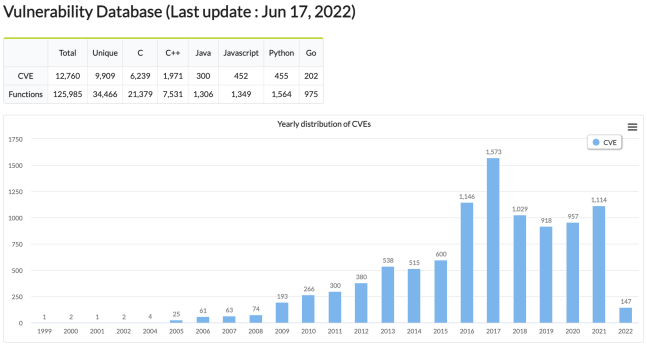


**FIGURE 16:** Vulnerability statistics by language and year on IoTcube (which is shown in the VDB menu of IoTcube).

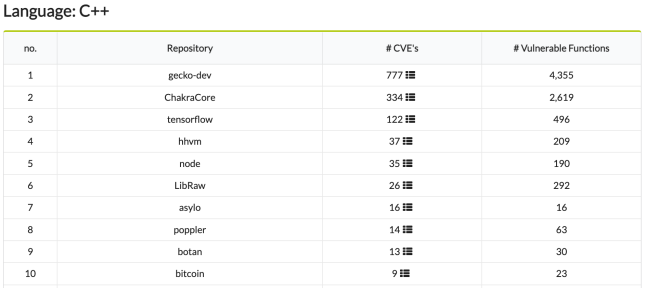


**FIGURE 17:** Vulnerability statistics by repository on IoTcube (which is shown in the VDB menu of IoTcube).

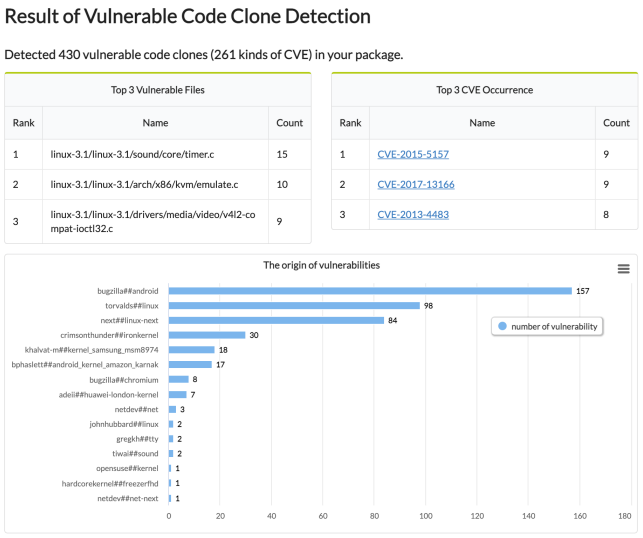


**FIGURE 18:** Result page on IoTcube. IoTcube detected 1,016 vulnerable code clones in the Linux software.

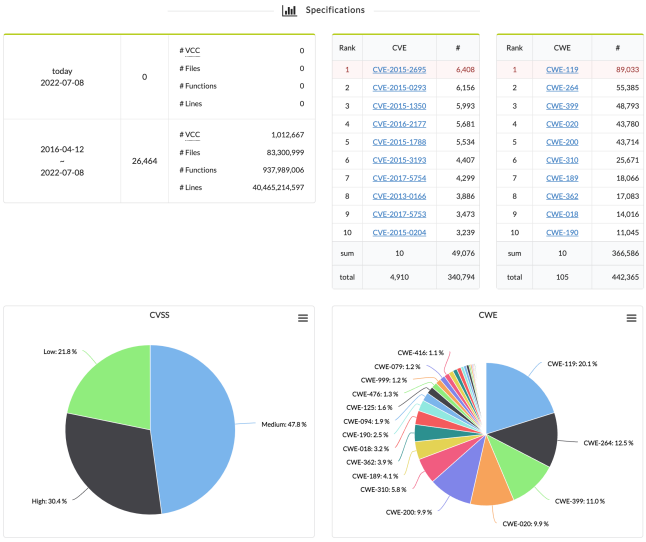


**FIGURE 19:** Statistics for IoTcube white box testing (which is shown in the Statistics menu of IoTcube).

To test vulnerable code clone detection technique in IoTcube, users first generate hash values of functions from the target software, using the "Hmark" tool, *i.e.*, the implementation tool of VUDDY [10]; the function hash values of the target software are embedded in the analysis file. Thereafter, users upload the analysis file to the IoTcube server. Finally, the IoTcube server gives the vulnerability detection reports in the platform (see Figure 18). We provide sufficient information for the detected vulnerabilities: the paths where the vulnerabilities were detected, the top CVE occurrences, the origin software of the detected vulnerabilities, distribution by year, CVSS, CWE, and tree view for vulnerable function paths.

Figure 19 depicts the statistics for the IoTcube white box testing (*i.e.*, vulnerable code clone detection, VUDDY [10]).

We confirmed that 26,464 users tested the tool, and approximately one million vulnerable code clones were detected from 83 million files since the release of IoTcube in 2016. We further confirmed that CWE-119 (*i.e.*, Buffer Overflow), CWE-264 (*i.e.*, Access Control Error), and CWE-399 (*i.e.*, Resource Management Error) were the top three CWE types.

## V. RELATED WORKS

In this section, we introduce a number of related works.

**Public vulnerability database.** There are various public vulnerability databases [4], [13]–[15], [19] that can freely search cybersecurity flaws assigned with a CVE ID. Such databases can be classified into national-managed databases and private company-managed databases. Many developers and security engineers use these databases because new CVE are updated via CVE MITRE, data feed [16] for the CVE list can be obtained from NVD, and CVE Details provides statistics on various criteria of CVE.

In addition, several companies including Snyk [19] and Mend [13] (known as WhiteSource) provide company-managed vulnerability databases. Compared to the national-managed databases, they further provide details about which version of the package manager is vulnerable to CVE.

However, such vulnerability databases do not fully provide information to address security threats caused by the vulnerabilities. Specifically, they do not essentially provide code-level security patches for all the vulnerabilities. Therefore, our approach provides vulnerable and patched code, making it effective in mitigating security threats at the source code level (we discuss this application in detail in Section VI).

**Collecting known security patches.** Several approaches (*e.g.*, [11], [12], [18], [21]–[23], [25]) have attempted to construct vulnerability databases by collecting security patches. However, most of the existing approaches only covered security patches that could be collected from repositories, especially Git (*e.g.*, [11], [12], [18], [21]–[23]), as we discussed throughout this paper. In addition, they only focused on collecting security patches with direct patch links, *i.e.*, missing many security patches with indirect and invisible patch links. Although VUDDY [10] and V0Finder [25] considered the collection method with an invisible patch link by searching for the keyword "CVE-20" in the commit messages, the method of collecting with only keyword features may cause many false positives; note that our approach overcome this problem by analyzing whether a patch contains "CVE-20" as well as control flow changes or security-sensitive API changes. In summary, existing approaches failed to collect a sufficient number of security patches owing to the presence of limited data sources and shallow scanning problems, while our proposed approach could collect much more security patches with higher patch collection accuracy.

## VI. DISCUSSION

**Storing vulnerable and patched codes.** To easily detect vulnerabilities, it is necessary to store security patches in a processed form. Therefore, we consider the unit for storing and the index to be used for vulnerability detection.

We determined that the function-level granularity unit is the most appropriate when using xVDB for vulnerability detection as the advantages of using the function unit (*i.e.*, high performance and scalability) have been already verified in existing approaches [1], [1], [10], [26], [27]. The method of extracting vulnerable and patches functions from security patches is a simple task that has been introduced in many studies. Using the index values of the security patch (*e.g.*, line #3 in Listing 1), we can access the file before (*resp.* after) applying the patch; we then extract the vulnerable (*resp.* patched) function from the accessed source file.

Although these collected vulnerable and patched functions can be directly utilized for vulnerability detection, most existing approaches use the hash value of the function the scalability (*e.g.*, [10], [25], [27]). With the extracted vulnerable and patched functions (normalized with removing comments, tabs, linefeed, and whitespaces, which are easy to change but do not affect program semantics), we created a hash index to be used for vulnerability detection. Here, three mechanisms can be performed for creating the hash index: generating hash values for (1) exact matching, (2) abstract matching, and (3) similarity matching.

(1) **Hash for exact matching:** A hash value extracted from the normalized function body of the original function. If the target program contains the same functions (*i.e.*, same characters) with a vulnerable function, it can be detected.

(2) **Hash for abstract matching:** A hash value extracted from the normalized function body of the abstracted function (every occurrence of the parameter, variable, and data types replace symbols [10]). Even if the target function slightly changes with the parameters, variables, and data types, the function can be detected.

(3) **Hash for similarity matching:** A hash value that can be used for similarity comparisons. If the target program contains the similar functions with a vulnerable function, it can be detected using this type of hash value (it can be processed using locality sensitive hashing algorithms).

When hash values of vulnerable and patched functions are generated through the aforementioned processes, security patches collected by xVDB can be easily utilized for vulnerability detection.

**Survey of the CVE patch distribution.** To know the status of the CVE patch distribution, we conducted two additional experiments: (1) the distribution of OSS and its patches for the affected software, and (2) the proportion of CVE vulnerabilities that can be covered by our methods.

To represent the distribution of OSS and the corresponding patches in NVD, we examined 220 affected software that reported more than 100 CVE. With 220 affected software, we classified them into two groups: OSS and commercial

software. To determine which software is OSS, we considered two criteria: (1) the case that can be extracted repository URLs from the NVD references and (2) the case that can extract repository URLs from external references (*e.g.*, Wikipedia [24], vendor website, and repositories). While the first criteria can be automatically identified, the cases of the second criteria were manually checked because they cannot be easily automated. Note that we considered software whose source codes are fully opened as the OSS, and a partially-open software was used as a commercial software. With these results, we measured the total number of CVE.

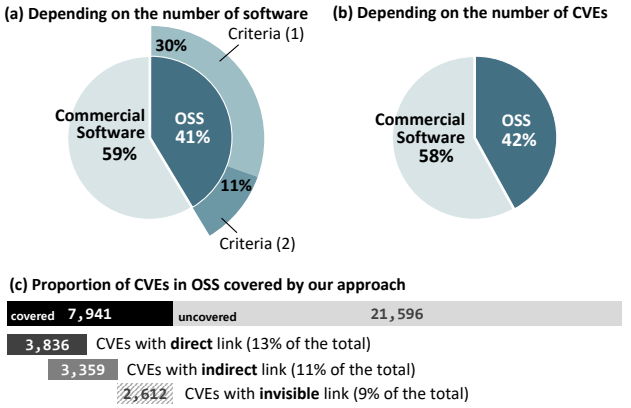


**FIGURE 20:** Status of the CVE patch distribution: (a) and (b) show the distribution of OSS and its patches for the affected software, and (c) depicts the proportion of vulnerabilities that can be covered by our methods.

Among the 220 software, we confirmed that 91 (*i.e.*, 41% of the total) were OSS; 67 software (*i.e.*, 30% of the total) were detected by the first criteria, and we manually identified 24 software (*i.e.*, 11% of the total) as open source from external references (see Figure 20 (a)). The cumulative number of CVE vulnerabilities reported by these 91 OSS projects was 29,537 (42% of all CVE vulnerabilities reported by the 220 affected software programs).

We targeted OSS to collect security patches because the vulnerabilities related to commercial software and hardware tend not to disclose their security patches. In addition, even in the case of OSS vulnerabilities, there are still many vulnerabilities that do not disclose security patches. Nevertheless, our approach could cover 7,941 CVE (27%) vulnerabilities among the 29,537 CVE vulnerabilities reported by the 91 OSS projects. In particular, the direct link collection method covered 3,836 CVE patches (13%), the indirect link collection method collected 3,359 CVE patches (11%), and the invisible link collection method gathered 2,612 CVE patches (9%). This is clearly the result of collecting more security patches compared to the existing approaches, and once again, this emphasizes the importance of collecting indirect and invisible patch links.

We failed to collect the remaining 21,596 CVE patches (73%) because they did not exhibit any hidden connectivity (defined in this paper) between the CVE info page and the CVE patch page. Instead of releasing security patches, their CVE info pages mainly suggest resolving vulnerabilities through OSS version updates. Since more and more security patches are being provided through direct links, the number of patches our approach can collect will also gradually increase. Nevertheless, developers need to provide a security patch at the time of reporting a vulnerability, so that other developers or security analysts can more clearly understand the cause and solution of the vulnerability.

## VII. CONCLUSION

As cybersecurity attacks aimed at OSS have exponentially grown, the need for a large-scale vulnerability database, which is effective in detecting, is also growing. In response, we constructed a large-scale patch database for known vulnerabilities, called xVDB. Our experimental results affirmed that our approach has a much higher coverage than existing techniques in terms of collecting security patches, as it can collect security patches at least four times more than those collected in existing approaches.

xVDB can be used to support vulnerability detection. The security patches collected in xVDB can be used to detect vulnerable code clones contained in real-world software, and this vulnerability detection approach has been serviced since 2016 through the IoTcube platform. Equipped with the patch information provided by xVDB, developers can address potential threats caused by propagated vulnerabilities, rendering a safe software ecosystem.

As a future extension, we will devise a patch collection method for security patches that do not have hidden connectivities between the CVE info page and the CVE patch page. For example, a patch for fixing a CVE vulnerability may not contain any hints (*e.g.*, CVE ID) about the vulnerability in its commit message. We are considering a method for collecting security patches based on the description information of the CVE info page.

## REFERENCES

[1] Mansour Ahmadi, Reza Mirzazade Farkhani, Ryan Williams, and Long Lu. Finding Bugs Using Your Own Code: Detecting Functionally-similar yet Inconsistent Code. In 30th USENIX Security Symposium (USENIX Security 21), 2021.
[2] Red Hat Bugzilla. Cairo issue tracker site, 2022. https://bugzilla.redhat.com/show_bug.cgi?id=1898396.
[3] Cairo. Cairo repository, 2022. https://gitlab.freedesktop.org/cairo/cairo/.
[4] CVE Details. CVE Details, 2022. https://www.cvedetails.com/.
[5] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. Identifying Open-Source License Violation and 1-day Security Risk at Large Scale. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS), pages 2169–2185, 2017.
[6] Felix Fischer, Konstantin Böttinger, Huang Xiao, Christian Stransky, Yasemin Acar, Michael Backes, and Sascha Fahl. Stack overflow considered harmful? the impact of copy&paste on android application security. In 2017 IEEE Symposium on Security and Privacy (SP), pages 121–136. IEEE, 2017.
[7] Hyunji Hong, Seunghoon Woo, and Heejo Lee. DICOS: Discovering Insecure Code Snippets from Stack Overflow Posts by Leveraging User Discussions. In Annual Computer Security Applications Conference (ACSAC), pages 194–206, 2021.
[8] IoTcube. IoTcube, 2022. https://iotcube.net/.

[9] Jiyong Jang, Abeer Agrawal, and David Brumley. Redebug: Finding unpatched code clones in entire os distributions. In 2012 IEEE Symposium on Security and Privacy (SP), pages 48–62. IEEE, 2012.
[10] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In 2017 IEEE Symposium on Security and Privacy (SP), pages 595–614. IEEE, 2017.
[11] Frank Li and Vern Paxson. A large-scale empirical study of security patches. In Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security (CCS), pages 2201–2215, 2017.
[12] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. SPIDER: Enabling Fast Patch Propagation in Related Software Repositories. In 2020 IEEE Symposium on Security and Privacy (SP), pages 1562–1579. IEEE, 2020.
[13] Mend. Mend Vulnerability DB, 2022. https://www.mend.io/vulnerability-database/.
[14] CVE MITRE. CVE MITRE, 2022. https://cve.mitre.org/.
[15] NVD. National Vulnerability Database, 2022. https://nvd.nist.gov/.
[16] NVD. NVD Data Feed, 2022. https://nvd.nist.gov/vuln/data-feeds.
[17] Stack Overflow. 2021 developer survey, 2022.
[18] Henning Perl, Sergej Dechand, Matthew Smith, Daniel Arp, Fabian Yamaguchi, Konrad Rieck, Sascha Fahl, and Yasemin Acar. VCCFinder: Finding Potential Vulnerabilities in Open-Source Projects to Assist Code Audits. In Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pages 426–437, 2015.
[19] Snyk. Snyk Vulnerability DB, 2022. https://security.snyk.io/vuln.
[20] Sonatype. State of the Software Supply Chain, 2021. https://www.sonatype.com/resources/state-of-the-software-supply-chain-2021.
[21] Xin Tan, Yuan Zhang, Chenyuan Mi, Jiajun Cao, Kun Sun, Yifan Lin, and Min Yang. Locating the Security Patches for Disclosed OSS Vulnerabilities with Vulnerability-Commit Correlation Ranking. In Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS), pages 3282–3299, 2021.
[22] Xinda Wang, Kun Sun, Archer Batcheller, and Sushil Jajodia. Detecting "0-Day" Vulnerability: An Empirical Study of Secret Security Patch in OSS. In 2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pages 485–492. IEEE, 2019.
[23] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. PatchDB: A Large-Scale Security Patch Dataset. In 2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pages 149–160. IEEE, 2021.
[24] Wikipedia. Wikipedia, 2022. https://www.wikipedia.org/.
[25] Seunghoon Woo, Dongwook Lee, Sunghan Park, Heejo Lee, and Sven Dietrich. V0Finder: Discovering the Correct Origin of Publicly Reported Software Vulnerabilities. In 30th USENIX Security Symposium (USENIX Security 21), pages 3041–3058, 2021.
[26] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse. In 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE), pages 860–872. IEEE, 2021.
[27] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. Mvp: Detecting vulnerabilities using patch-enhanced vulnerability signatures. In 29th USENIX Security Symposium (USENIX Security 20), pages 1165–1182, 2020.

HYUNJI HONG received the B.S. degree in Computer Science and Engineering from Hanshin University, Gyeonggi-do, South Korea, in 2020. She is a M.S. candidate in the Department of Computer Science and Engineering, Korea University, Seoul, South Korea. Her research interests include software security, vulnerability detection, vulnerability analysis.

SEUNGHOON WOO received the B.S. degree in Computer Science and Engineering from Korea University, Seoul, South Korea, in 2016. He is a Ph.D. candidate in the Department of Computer Science and Engineering, Korea University. His research interests include software security, vulnerability detection, and software composition analysis. He has published papers on software security and software engineering in top conferences such as S&P, USENIX Security, and ICSE.

EUNJIN CHOI received the B.S. degree in Computer Science and Engineering from Inha University, in 2020, and M.S. degree in the Department of Computer Science and Engineering from Korea University, Seoul, South Korea, in 2022. Her research interests include vulnerability detection, vulnerability analysis, and digital forensics.

JIHYUN CHOI is an undergraduate student majoring in Computer Science and Engineering from Korea University, Seoul, South Korea. Her research interests include open source software security and vulnerability analysis.

HEEJO LEE (Member, IEEE) is the Professor in the Department of Computer Science and Engineering at Korea University, and the director of Center for Software Security and Assurance (CSSA). He received his B.S., M.S., and Ph.D. degree in Computer Science and Engineering from POSTECH. Before joining Korea University, he served as a CTO at AhnLab Inc. from 2001 to 2003, and as a Post-doctorate Researcher at Purdue University from 2000 to 2001. He is an Editor of the Journal of Communications and Networks, and the International Journal of Network Management. He is a founding member and co-CEO of IOTCUBE Inc.