Clovery: Identifying Affected Versions in C/C++ Public Security Vulnerability Reports

Duyeong Kim[1][0009−0007−5426−9661], Jimin Kang[1][0009−0008−7940−5962], Yeonhee Kim[1][0009−0003−5676−5811], Seunghoon Woo[1*][0000−0002−5455−0804], and Heejo Lee[1*][0000−0002−5831−0787]

Korea University, Seoul, South Korea
{duyeong,ziminii,kyh0502,seunghoonwoo,heejo}@korea.ac.kr

Abstract. We present Clovery, a precise approach for detecting vulnerability-affected versions through code-level semantic analysis. Precise management of vulnerability-affected versions is crucial for mitigating risks from vulnerability propagation. However, identifying these versions is challenging, requiring deep understanding of both the syntax and semantics of vulnerable code. Existing approaches often rely on broad assumptions, causing false alarms. Public security reports frequently misclassify dangerous versions as safe and *vice versa*, further exacerbating the problem. To overcome these limitations, Clovery uses a technique called *function-lineage tracking*. It focuses on vulnerability-related code lines and uses semantic pairs to capture dependencies. By tracking function histories via semantic pairs, Clovery precisely identifies versions affected by vulnerabilities. On 1,502 NVD CVEs, Clovery found potential CPE issues in 65.65% of cases, indicating risks to vulnerability management. Notably, Clovery achieved 97.43% F1 in identifying affected versions, significantly outperforming existing approaches, which achieved 71.16% and 84.23% F1 scores. These results suggest that Clovery can improve vulnerability response and software supply-chain security.

1 Introduction

The widespread adoption of open-source software (OSS) has accelerated software development by enabling code reuse through public repositories [2,1,25,29]. However, library reuse in OSS introduces security concerns such as vulnerability propagation [13,11]. To mitigate these risks, vulnerabilities are disclosed as Common Vulnerabilities and Exposures (CVEs) and documented in the National Vulnerability Database (NVD) [3], which includes affected software versions using the Common Platform Enumeration (CPE). This information plays a crucial role in helping developers identify and remediate potential threats [22,5].

Despite its role, affected version information is often inaccurate [4,24,8,26]. The NVD may over-approximate affected ranges, treating all preceding versions as vulnerable. Conversely, a vulnerability may exist in a version, but the NVD incorrectly indicates it is safe. Furthermore, the NVD often fails to account for

* To whom all correspondence should be addressed.

parallel development (*i.e.*, two or more release lines maintained concurrently), leading to inaccuracies in tracking affected versions (see Section 2.2).

Existing approaches. Existing approaches to identifying affected versions have limitations due to (1) overly simplistic assumptions and (2) exclusive focus on patch syntax. A key assumption is that code critical to vulnerabilities is always deleted in patches. For example, `VOFinder` [24] assumes that only deleted code lines are critical and therefore cannot handle cases where patches only add new lines. VISION [26] assumes detailed descriptions of vulnerabilities (*e.g.*, code lines causing the vulnerability) are provided in vulnerability reports, making it difficult to comprehensively examine CVEs. SZZ techniques (*e.g.*, [20,4]) identify commits where a bug was introduced; however, for identifying vulnerability-affected versions this approach is less effective because both the commit introducing the vulnerability and the one removing it are important.

Our approach. To address these shortcomings, we present CLOVERY (Code cLOne based vulnerability-affected VErsion discoveRY), a precise approach for identifying versions affected by vulnerabilities. CLOVERY differs from existing approaches by (1) focusing on core code lines closely tied to vulnerabilities and (2) leveraging semantic pairs to capture code dependencies.

Given a CVE ID, CLOVERY collects the security patch and codebase of the affected software. It reconstructs vulnerable and patched functions from the security patch and employs *function-lineage tracking* to capture semantic relationships as *semantic pairs*. CLOVERY considers a version vulnerable if it contains functions with vulnerable semantic pairs and few or no patch semantic pairs, and aggregates these results to determine the full range of affected versions.

To evaluate CLOVERY, we collected 4,060 C/C++ CVEs from the NVD and selected 1,502 for evaluation (selection criteria detailed in Section 5.1). CLOVERY identified potential CPE inconsistencies in 65.65% of these CVEs (see Section 5.2). One major cause of such errors is the lack of mechanisms to account for new branches or versions introduced after a CVE is published. Despite this challenge, CLOVERY outperformed existing approaches in identifying vulnerability-affected versions, achieving a 97.43% F1 score compared to 71.16% and 84.23% F1 scores by existing approaches [24,20]. CLOVERY also analyzes the types and root causes of errors in public security reports. Our findings indicate that affected version information in the NVD requires ongoing updates, and CLOVERY provides an effective automated solution.

Contributions. We summarize our contributions below.

- We propose CLOVERY, a precise approach for identifying vulnerability-affected versions. The key technical contribution is *function-lineage tracking*, which considers pairs of code lines with semantic relationships.
- We found that many public security reports incorrectly specify vulnerability -affected versions (65.65% of cases in our experimental setup).
- CLOVERY achieved 96.27% precision and 98.62% recall in identifying vulnerability-affected versions, outperforming existing approaches.
- Source code and datasets related to CLOVERY will be publicly available at publication.

2 Motivation

2.1 Basic terminology

Version-related terms are defined as follows:

- **Version**: A release of a software program, identified by version information specified in CPEs and by tags in the corresponding GitHub repositories.
- **Vulnerability-affected versions**: Versions that are affected by a vulnerability in the software where the vulnerability originated, including versions that may be indirectly affected.
- **Vulnerable software information in public vulnerability reports**: Affected-version metadata reported in public security reports (e.g., CPEs), which this paper aims to validate.

2.2 Motivating example

Listing 1.1: CVE-2018-19210: a patch snippet showing the fix is missing.

```
// CPE (reported vulnerable): cpe:2.3:a:libtiff:libtiff:4.0.9:*:*:*:*:*:*:*
// CPE (reported fixed) : cpe:2.3:a:libtiff:libtiff:4.0.10:*:*:*:*:*:*:*
// Commit (patch) : d0a842c5dbad2609aed43c701a12ed12461d3405
// Path : libtiff/tif_dir.c
static int _TIFFVSetField(...) {
...
   td->td_smaxsamplevalue = NULL;
  }
// --- LibTIFF 4.0.10 (unpatched): expected fix is missing ---
- if( td->td_transferfunction[0] != NULL && (v - td->td_extrasamples > 1) &&
-    !(td->td_samplesperpixel - td->td_extrasamples > 1)) {...}
// --- Patch commit d0a842c5...: fix is present ---
+ if( td->td_transferfunction[0] != NULL && (v - td->td_extrasamples > 1) &&
+    !(td->td_samplesperpixel - td->td_extrasamples > 1)) {...}
 }
 td->td_samplesperpixel = (uint16) v;
}
```

We present a `LibTIFF` case where affected version information in a public security report leads to misclassification of vulnerable and safe versions.

`LibTIFF`[1] is widely used for reading and writing Tagged Image File Format (TIFF) files. CVE-2018-19210 reports a NULL pointer dereference and indicates that the issue was patched in 4.0.10. However, we found that the fix is missing in `LibTIFF` 4.0.10 and was applied in a later version (see Listing 1.1). We confirmed that this vulnerability was reproduced in `LibTIFF` 4.0.10 using a known Proof of Concept (PoC). Developers often choose to partially update or backport patch code, due to compatibility or dependency issues [25,16]. In this context, inaccurate affected version information can misguide partial updates and negatively impact overall system security.

3 Methodology of Clovery

In this section, we present the methodology of CLOVERY, which precisely identifies affected versions of C/C++ vulnerabilities. The key distinguishing feature of CLOVERY is its consideration of both the syntax and semantics of code lines by using semantic pairs and function-lineage tracking.

[1] https://gitlab.com/libtiff/libtiff

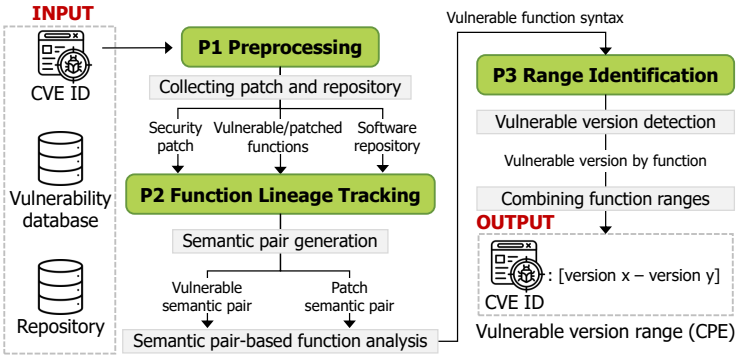


Fig. 1: High-level overview of Clovery.

Figure 1 illustrates the workflow of Clovery, which comprises the three phases: *preprocessing* (**P1**), *function-lineage tracking* (**P2**), and *range identification* (**P3**). In **P1**, Clovery collects the security patch and the corresponding repository, and extracts vulnerable and patched functions. In **P2**, Clovery extracts semantic pairs and tracks target functions over the commit history to identify vulnerable syntaxes. In **P3**, Clovery aggregates function-level results to identify the affected-version range for the input CVE.

3.1 Preprocessing phase (P1)

Security patch and repository collection. Given a CVE, Clovery collects security patches by checking CVE reference links in the NVD. If a reference link includes a GitHub commit URL, Clovery identifies it as a security patch and collects it (see Section 4). Clovery then collects the repository associated with the patch URL [23,22].

Vulnerable and patched function extraction. Subsequently, Clovery extracts vulnerable and patched functions using the collected security patches and repositories. Because Clovery collects security patches from GitHub commits, it extracts the modified files and recovers the functions containing the changed lines. Unlike approaches (*e.g.*, [24]) that focus only on added or deleted lines, Clovery extracts full functions to capture lines dependent on the patch changes.

3.2 Function-lineage tracking phase (P2)

Using the security patch, repository, and functions from P1, Clovery performs *function-lineage tracking*. This phase consists of four steps: core code line extraction (**S1**), semantic pair configuration (**S2**), commit history extraction (**S3**), and vulnerable syntax identification (**S4**).

S1: Core code line extraction. Clovery extracts two types of core lines: **essential lines**, which are changed in the security patch (*i.e.*, added or deleted code lines), and **dependent lines**, which are within the same function and have control or data dependency on the essential lines.

Essential lines are code segments that play a critical role in the manifestation of vulnerabilities [13]. However, as the codebase evolves, these lines may

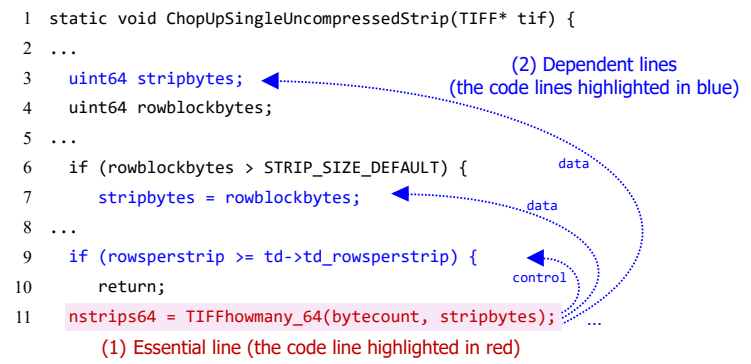


Fig. 2: Illustration of identifying essential and dependent lines to generate semantic pairs.

change [27,23], making it necessary to perform additional semantic analysis to assess the impact of such changes. To address this, CLOVERY also considers dependent code lines those that support or rely on the behavior of essential lines and may influence vulnerabilities [27,23]. By pairing semantically related lines, CLOVERY can effectively identify vulnerable code clones and determine affected versions. Specifically, CLOVERY first treats the code lines *deleted* in the patch as essential, then identifies their dependent lines within the *vulnerable* function. It then performs the same analysis for the *added* lines in the *patched* function.

S2: Semantic pair configuration. CLOVERY focuses on code line pairs (called *semantic pairs*) that have a semantic relationship within a function. CLOVERY considers two types of semantic pairs: *vulnerable semantic pairs* extracted from the vulnerable function and *patch semantic pairs* extracted from the patched function. A semantic pair (p) consists of an essential line (l_e) and a dependent line (l_d) within the same function: $p = (l_e, l_d)$. For each essential line, CLOVERY generates semantic pairs with all identified dependent lines.

Figure 2 illustrates the extraction process for vulnerable semantic pairs. Code lines #3, #7, and #9 are marked as dependent due to their data dependency on line #11 (*e.g.*, dependency on the variable `stripbytes`). This yields three vulnerable semantic pairs: line `(#11, #3)`, `(#11, #7)`, and `(#11, #9)`. By applying this process to all essential and dependent lines (for both vulnerable and patched functions), CLOVERY extracts semantic pairs for each essential line.

The issue is that the syntax of added and deleted lines may be identical, and a dependent line can be shared across vulnerable and patch semantic pairs, which can hinder identification. To address this, CLOVERY ignores identical add-delete lines and removes duplicate dependent lines from patch semantic pairs.

S3: Commit history extraction. Next, CLOVERY extracts the change history of the functions in which vulnerabilities exist. Because CLOVERY collects the codebase of the software with vulnerabilities from GitHub (see Section 3.1), this can be easily accomplished using git commands (*e.g.*, using "`git log`"). CLOVERY sets the software codebase corresponding to the vulnerability to the

latest version, and extracts the commit history of the source files containing vulnerable functions (details are provided in Section 4).

S4: Vulnerable syntax identification. Using semantic pairs and the extracted commit history, CLOVERY identifies vulnerable syntaxes in target functions. If function names change, CLOVERY tracks target functions from the patch commit in both directions of the file history.

1. **Set the reference point.** CLOVERY uses the patch commit as the reference point because it includes the target function names extracted in P1.
2. **Function tracking.** CLOVERY traverses the history in both directions and keeps only commits where the target functions change.

To reduce false alarms from non-semantic changes and trivial code lines, CLOVERY applies normalization and filters short lines. CLOVERY ignores comment lines, converts characters to lowercase, and removes whitespace. CLOVERY also disregards code lines with 15 or fewer characters after normalization [23].

CLOVERY determined that the target function has a vulnerable syntax if it contains a sufficient number of vulnerable semantic pairs while including a few (or no) patch semantic pairs. Loose matching was employed to address the diversity of the function syntax. Based on this idea, CLOVERY considers the following two types of similarities: *vulnerability similarity* (Φ_v) and *patch similarity* (Φ_p).

Let P_v and P_p represent vulnerable and patch semantic pairs, respectively, and let f_i denote the target function.

$$\Phi_v = \frac{|\{(x,y) \in P_v \mid x \in f_i \wedge y \in f_i\}|}{|P_v|}, \quad \Phi_p = \frac{|\{(x,y) \in P_p \mid x \in f_i \wedge y \in f_i\}|}{|P_p|}$$

Similarity is defined as the ratio of semantic pairs in which both the essential and dependent lines exist. We introduce two thresholds, θ_v and θ_p, and use them to determine whether the syntax of a target function f_i should be considered vulnerable. CLOVERY determines that f_i is vulnerable if $\Phi_v > \theta_v$ and $\Phi_p < \theta_p$.

If no code lines are added (*resp.* deleted) in the patch, CLOVERY considers only $\Phi_v > \theta_v$ (*resp.* $\Phi_p < \theta_p$). Because CLOVERY considers semantic pairs rather than isolated lines, it can handle patches with structural changes and code dependencies that are missed by syntax-only approaches [24,26]. Compared to VISION [26], CLOVERY does not rely on additional assumptions from descriptions or vulnerability patches, enabling it to identify versions affected by vulnerabilities much more comprehensively.

3.3 Range identification phase (P3)

Finally, CLOVERY identifies the affected-version range for the input CVE by aggregating function-level results from P2. For each version, CLOVERY checks whether any target function instance is classified as vulnerable; if so, CLOVERY marks the version as affected.

Let F_v be the set of vulnerable syntaxes for the target function instances, v_i represent each version, and f_i denote the syntax of the target function in v_i.

Vulnerability affected version identification

• CLOVERY determines that v_i is the vulnerability affected version when f_i has the same syntax as f_j contained in F_v.

Some may consider directly comparing semantic pairs with f_i instead of extracting vulnerable function syntaxes. However, as mentioned earlier, this approach may fail to correctly extract f_i from v_i, particularly when the names of the target functions have changed. Therefore, CLOVERY identifies versions affected by vulnerabilities based on function-lineage tracking. Note that since a patch can modify multiple functions, CLOVERY evaluates each function and aggregates the results across modified functions.

4 Implementation of Clovery

Architecture of Clovery. CLOVERY comprises three modules: a *dataset collector*, *function-lineage tracker* and *vulnerable range identifier*. All process modules were implemented in approximately 2,900 lines of Python code, excluding external libraries (*e.g.*, function parsers).

CVE patch collection. We obtained security patches using a method similar to those in previous studies (*e.g.*, [10,14,23]). We collected security patches by crawling `Git` commit URLs referenced in NVD CVE entries. Consequently, we collected 4,060 C/C++ security patches from the NVD (as of August 2024).

We then extracted the vulnerable and patched functions from the security patches using techniques from previous studies (*e.g.*,[13,23]). Using file indices before and after patch application, we extracted functions containing modified code. To extract functions, we used Universal Ctags[6], a fast and accurate function parser. In total, we extracted 10,359 vulnerable–patched function pairs. CLOVERY then leverages the Joern parser [28] to analyze extracted functions, identifying essential and dependent lines to construct semantic pairs.

Software repository and commit history collection. We collected software repositories corresponding to the security patches obtained from GitHub. Consequently, we collected 783 software repositories with a total of 148,970 versions. For each target function, CLOVERY tracks the change history by collecting commit histories of the files containing the function using `Git` commands.

```
git log --follow --diff-filter=AMD --name-status --pretty=format:"%H
                        %ci" -- <file_path>
```

Using the extracted commit history, CLOVERY identifies commits that include changes to the target function and extracts the corresponding function versions. In addition, CLOVERY retrieves function-level code changes in `diff` format using the following command.

```
git log -1 <commit_id> -L:<function_name>:<path>
```

If the target function was renamed, we updated its name to ensure continuous history tracking.

Table 1: Accuracy of NVD, `VOFinder`, `SEM-SZZ`, and Clovery in identifying vulnerability-affected versions. Clovery achieved the best overall precision and recall across all three validation methods.

Validation method	#CVEs	NVD			`VOFinder`[24]			`SEM-SZZ`[20]			**Clovery**		
		Precision	Recall	F1 score	Precision	Recall	F1 score	Precision	Recall	F1 score	Precision	Recall	F1 score
Reproduction	103	0.8627	0.8712	0.8669	**0.9973**	0.6645	0.7976	0.9421	0.8574	0.8978	0.8837	**0.9759**	**0.9275**
Author sites	54	0.8913	0.8612	**0.8759**	0.9129	0.5158	0.6592	**0.9570**	0.5108	0.6661	0.7302	**0.9245**	0.8160
Code review	1,345	0.8126	0.8566	0.8340	0.9450	0.5647	0.7070	0.9460	0.7644	0.8456	**0.9781**	**0.9894**	**0.9837**
Total	1,502	0.8189	0.8577	0.8379	0.9474	0.5698	0.7116	0.9484	0.7576	0.8423	**0.9627**	**0.9862**	**0.9743**

5 Evaluation

We evaluate Clovery based on the following three research questions.

- **RQ1: Accuracy.** How precisely does Clovery identify versions affected by vulnerabilities compared with existing approaches and CPE in NVD? What are representative examples?
- **RQ2: Findings.** Based on the results from Clovery, how much difference is observed in CPE? What are the types and causes of these differences?
- **RQ3: Performance.** How does Clovery perform in terms of runtime when identifying the versions affected by vulnerabilities?

We ran Clovery on a machine with a GNU/Linux 6.5.0-44-generic x86_64, Intel (R) Core (TM) i7-13700 @ 5.70GHz, 62GB RAM, and 1.8TB SSD.

5.1 Accuracy

Accuracy measurement targets. Out of 4,060 collected CVEs, Clovery extracted core lines for 4,054 CVEs (six failures due to external-library errors). To analyze affected versions, we excluded vulnerabilities reported in GitHub repositories that do not specify a version. In this process, 1,878 CVEs were removed. Among the remaining 2,176 CVEs, we further excluded cases where the GitHub repository and version names are not meaningfully aligned with the CPE software identity, because Clovery does not aim to resolve CPE–repository name mismatches. Consequently, Clovery analyzed *1,502 CVEs*.

The problem is that there is no definitive ground truth for vulnerability-affected versions. Hence, we manually validated the results using three methods: *reproduction*, *author sites*, and *code review* (*i.e.*, referring to `VOFinder` [24]). First, we attempt to trigger the vulnerability in a target version when exploit information is available (*reproduction*). We also refer to reports from the repository's website or blogs where the vulnerability was reported and analyze the reports (*author sites*). If none of the aforementioned information was available, we examined the source code to identify vulnerable functions and determine whether the version was affected (*code review*).

Four analysts conducted the validation (one with 10+ years of security code-review experience and three with 5+ years of research experience). During code reviews, analysts did not assume all code modifications to be security-relevant. Instead, they carefully examined each patch in the context of the associated CVE, referring to the CVE description, commit message, and surrounding source

code. The review focused on determining whether the modified code is directly associated with the occurrence or mitigation of the reported vulnerability, particularly by analyzing whether the relevant data or control flow remains consistent with the vulnerability type. This manual validation enables semantic understanding and contextual interpretation that are often difficult to capture through automated analysis alone.

We classify results into TP (vulnerable identified as vulnerable), FP (safe identified as vulnerable), and FN (vulnerable identified as safe). We use macro-accuracy to avoid bias from uneven version counts per CVE. Macro-precision (P) and macro-recall (R) are defined as $P = \frac{1}{N}\sum_i \frac{TP_i}{TP_i+FP_i}$ and $R = \frac{1}{N}\sum_i \frac{TP_i}{TP_i+FN_i}$, where N is the number of CVEs. We then compute the F1 score (F): $F = (2 * P * R)/(P + R)$. We conducted the experiments by setting θ_v and θ_p to 0.5, respectively (see Section 3.2). The experiment on threshold sensitivity is introduced at the end of Section 5.1.

Comparison targets. We evaluated CLOVERY against the NVD (CPE) and compared it with `VOFinder` [24] and `SEM-SZZ` [20]. `AFV` [18] was excluded because it targets web-application vulnerabilities. VISION [26] could not be directly compared because its tool/source code were unavailable despite our request.

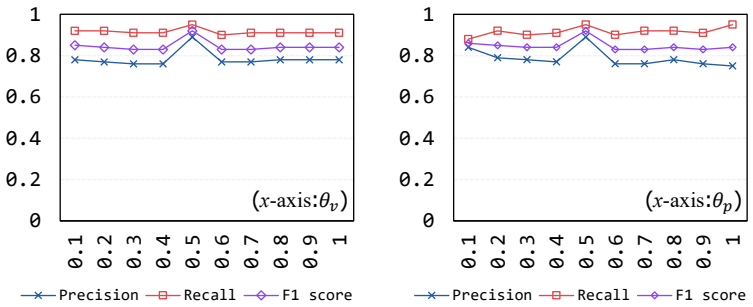
Result analysis. Table 1 summarizes the results on 1,502 CVEs. CLOVERY achieved 97.43% F1 (96.27% precision and 98.62% recall), thereby outperforming `VOFinder` (71.16% F1) and `SEM-SZZ` (84.23% F1). The NVD also showed non-trivial inaccuracies, which we analyze in Section 5.2.

`VOFinder` tends to miss vulnerabilities when patches mainly add lines or when vulnerable logic is structurally altered, as it relies on deleted-line heuristics and syntactic matching. `SEM-SZZ` handles added-line cases, but its SZZ-style linkage between inducing and fixing commits can be ambiguous across parallel branches, which may distort affected ranges.

In contrast, CLOVERY achieved higher accuracy compared to the existing tools. We requested NVD modifications for 20 reproduced CVEs; as of October 2025, four[2] of them were updated based on our requests. However, it produced several false alarms. For FPs, there were cases where semantic pairs related to the vulnerability were present, but the vulnerability did not manifest (*i.e.*, validated by *reproduction*); these could be TPs, but we considered versions that failed to trigger the vulnerability to be FPs in reproduction. In addition, there were cases where a patch added code lines, and in the subsequent commit, all the added lines were changed. In such cases, CLOVERY failed to detect functions containing the added code lines from the patch, leading to the misidentification of safe versions as vulnerable (*i.e.*, FPs). Most of the FNs in CLOVERY were due to the presence of code that could trigger the vulnerability but was not included in the semantic pairs extracted by CLOVERY (*e.g.*, code lines that cause vulnerabilities but are not related to the lines modified in the patch).

Threshold sensitivity. We measured sensitivity by fixing one threshold and varying the other from 0.1 to 1.0 (step 0.1) on CVEs validated via reproduction.

[2] CVE-2016-9388, CVE-2020-24370, CVE-2021-45985, and CVE-2023-50246



(a) Measuring accuracy while varying θ_v (with θ_p set to 0.5). **(b) Measuring accuracy while varying θ_p (with θ_v set to 0.5).**

Fig. 3: Results of accuracy measurements with one threshold fixed and the other threshold varied.

Figure 3 shows that accuracy is stable across θ, and the best results were achieved at $\theta_v = \theta_p = 0.5$; thus, we use 0.5 in subsequent experiments.

5.2 Findings

We then examine the discrepancies between the results of CLOVERY and the NVD (Table 1). The detection results of CLOVERY matched those of the NVD for only 343 out of 1,502 target CVEs (22.84%). Among the remaining 1,159 CVEs, CLOVERY showed 87 incorrect cases, while both CLOVERY and NVD were incorrect in 86 cases. We focus on 986 CVEs (65.65%) where CLOVERY provides correct affected versions but the NVD does not, and summarize their range-difference types in Figure 4 and Table 2.

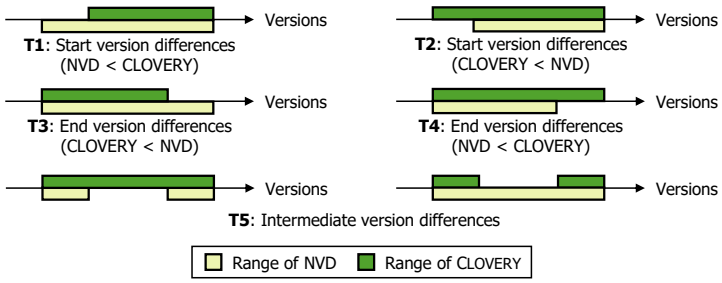


Fig. 4: Range difference types between Clovery and NVD.

We observed that the NVD yields more FPs (T1 and T3) than FNs (T2 and T4). In 903 CVEs (91.58%), the start version in the NVD differed from that identified by CLOVERY (T1 or T2). Most start-version mismatches were T1, indicating NVD overclaims: once a vulnerability is observed in a version, the NVD often marks all earlier versions as vulnerable. T2 cases (325 CVEs) mainly stemmed from NVD excluding very old versions despite evidence of vulnerability.

We also found 551 CVEs (55.88%) with end-version discrepancies (T3 or T4). T4 cases often occurred when NVD fixed ranges reflected only the reproduced version, omitting later vulnerable releases. T3 cases (194 CVEs) typically

Table 2: Distribution of range-difference types between Clovery and the NVD (duplicates allowed per CVE).

Type	Description	#CVEs	Ratio
T1	Start `diff`. (NVD < CLOVERY)	578	58.62% (578/986)
T2	Start `diff`. (CLOVERY < NVD)	325	32.96% (325/986)
T3	End `diff`. (CLOVERY < NVD)	194	19.68% (194/986)
T4	End `diff`. (NVD < CLOVERY)	357	36.21% (357/986)
T5	Intermediate `diff`.	184	18.66% (184/986)

arose after disclosure, when some versions were patched via backporting or other mitigations but the NVD range was not updated.

Finally, there were 184 CVEs where differences were observed in the intermediate versions (*i.e.*, T5). In most cases, similar to the T3 cases, these differences occurred because developers applied security patches to some versions through backporting after the CVE was reported. While there are some instances where the differences arose because CLOVERY failed to correctly identify the vulnerable versions, as confirmed in Section 5.1, these were very few. Among several cases we reported, CVE-2023-50246 was initially listed as affecting one version, but we confirmed that later versions (1.7-rc1 and 1.7-rc2) were also vulnerable. The findings indicate that the NVD often provides inaccurate affected version data. While not obligated to ensure high accuracy, the NVD is commonly used by developers for vulnerability response. Therefore, validating this information is critical for software security. CLOVERY presents an effective solution by automatically identifying affected versions at the time of vulnerability disclosure.

5.3 Performance

We measured the detection time of CLOVERY per CVE, excluding dataset collection, on 2,176 CVEs. CLOVERY analyzed a CVE in 295 s on average (median 72 s); 1,291 CVEs (about 60%) finished within 100 s. Runtime mainly increased with (i) the number of commits in the vulnerable file, (ii) the number of modified vulnerable functions, and (iii) the number of semantic pairs to compare, but is practical given that CLOVERY examines the full history of vulnerable functions.

6 Discussion

Limitations. CLOVERY requires (i) a security patch and (ii) the corresponding repository with commit history. CLOVERY may fail when a patch mainly reorders lines such that vulnerable and patched essential lines become syntactically identical; this is rare in our setup (< 0.05%) but can cause false alarms. We leave handling such cases (e.g., position-aware matching of essential lines) for future work. Although we implemented CLOVERY for C/C++, the methodology is largely language-agnostic and can be extended to other languages.

Threats to validity. Our evaluation covers 1,502 CVEs, which may not represent all vulnerability patterns. We restricted targets to cases where CPE identities and repository versions are reasonably mappable, which may bias the dataset toward well-maintained projects. Furthermore, due to the lack of ground truth,

we manually analyzed CLOVERY's results. While most cases were clearly resolved, and many vulnerabilities were verified through reproduction, the outcomes may not be perfect. Finally, analyst expertise was not uniform, and a senior reviewer may have influenced decisions, potentially introducing bias.

7 Related work

Prior studies have reported that NVD CPE affected-version data are often inaccurate (*e.g.*, [8,4,24,7,19,18,21]). Among them, `SEM-SZZ` [20] links a patch to its inducing commit and marks the versions between them as vulnerable, while `VOFinder` [24] refines CPE by locating where a vulnerability originated; other efforts target web vulnerabilities or specific languages (*e.g.*, [18,19]). However, these approaches remain largely patch-centric: they often rely on changed lines (especially deleted ones) and do not model semantic dependencies between code segments, which limits robustness under code evolution (see Section 5).

In addition, vulnerable code clone detection approaches have been widely studied (*e.g.*, [13,27,19,23,22,15,30]). Recent systems such as MOVERY [23] and V1SCAN [22] detect propagated clones using patch-derived signals, but they still tend to be sensitive to patch-line choices (e.g., deleted-line bias) and typically assume that affected-version metadata are reliable. CLOVERY follows the patch-based paradigm but differs by encoding semantic dependencies as pairs, enabling version-range identification without relying on CPE correctness; the resulting ranges can also improve downstream clone-detection accuracy. Code-lineage tracking has also been explored in various forms, but it does not address the key requirement in our setting (*e.g.*, [12,9,17]): distinguishing vulnerable and patched semantics to identify affected version ranges.

8 Conclusion

Vulnerability-affected versions provided in public vulnerability reports are often inaccurate, which can complicate vulnerability resolution. We introduce CLOVERY, a semantic pair-based function lineage tracking technique that integrates code syntax and semantics to accurately identify affected versions for CVE vulnerabilities. With CLOVERY, developers will be better equipped to respond to vulnerabilities, leading to improved supply chain security.

Acknowledgment

We thank the anonymous reviewers for their insightful comments to improve the quality of the paper. This work was supported by the Institute of Information & Communications Technology Planning & Evaluation (IITP), grant funded by the Korea government (MSIT) (No. RS-2024-00440780 Development of Automated SBOM and VEX Verification Technologies for Securing Software Supply Chains), ICT Creative Consilience Program (IITP-2026-RS-2020-II201819, 10%), and the National Research Foundation of Korea(NRF) grant funded by the Korea government(MSIT) (RS-2025-00517788, Research on Intelligent SBOM Generation and Automated Vulnerability Analysis through Multi-level Code Analysis).

References

1. Octoverse: The state of open source and rise of AI in 2023 - The GitHub Blog. https://github.blog/news-insights/research/the-state-of-open-source-and-ai/, (Accessed on 08/23/2024)
2. Open Source Security & Risk Analysis Report (OSSRA) — Synopsys. https://www.synopsys.com/software-integrity/resources/analyst-reports/open-source-security-risk-analysis.html#UXexecutiveSummary, (Accessed on 08/23/2024)
3. NVD. National Vulnerability Database (2024), https://nvd.nist.gov/
4. Bao, L., Xia, X., Hassan, A.E., Yang, X.: V-SZZ: Automatic Identification of Version Ranges Affected by CVE Vulnerabilities. In: 2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE). pp. 2352–2364 (2022). https://doi.org/10.1145/3510003.3510113
5. Choi, Y., Woo, S.: TIVER: Identifying Adaptive Versions of C/C++ Third-Party Open-Source Components Using a Code Clustering Technique. In: Proceedings of the 47th International Conference on Software Engineering (ICSE). IEEE (2025)
6. Ctags: Universal Ctags (2024), https://github.com/universal-ctags/ctags
7. Dai, J., Zhang, Y., Xu, H., Lyu, H., Wu, Z., Xing, X., Yang, M.: Facilitating Vulnerability Assessment through PoC Migration. In: Proceedings of the ACM Conference on Computer and Communications Security. pp. 3300–3317 (2021)
8. Dong, Y., Guo, W., Chen, Y., Xing, X., Zhang, Y., Wang, G.: Towards the Detection of Inconsistencies in Public Security Vulnerability Reports. In: 28th USENIX security symposium (USENIX Security 19). pp. 869–885 (2019)
9. Gharehyazie, M., Ray, B., Filkov, V.: Some from Here, Some from There: Cross-Project Code Reuse in GitHub. In: 2017 IEEE/ACM 14th International Conference on Mining Software Repositories. pp. 291–301 (2017)
10. Hong, H., Woo, S., Choi, E., Choi, J., Lee, H.: xVDB: A High-Coverage Approach for Constructing a Vulnerability Database. IEEE Access **10**, 85050–85063 (2022). https://doi.org/10.1109/ACCESS.2022.3197786
11. Jang, J., Agrawal, A., Brumley, D.: ReDeBug: Finding Unpatched Code Clones in Entire OS Distributions. In: 2012 IEEE Symposium on Security and Privacy. pp. 48–62 (2012). https://doi.org/10.1109/SP.2012.13
12. Kanda, T., Ishio, T., Inoue, K.: Approximating the Evolution History of Software from Source Code. IEICE Transactions on Information and Systems **98**(6), 1185–1193 (2015)
13. Kim, S., Woo, S., Lee, H., Oh, H.: VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In: Proceedings of the 38th IEEE Symposium on Security and Privacy (SP). pp. 595–614 (2017)
14. Li, F., Paxson, V.: A Large-Scale Empirical Study of Security Patches. In: Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security. pp. 2201–2215 (2017)
15. Li, Z., Zou, D., Xu, S., Jin, H., Qi, H., Hu, J.: Vulpecker: an automated vulnerability detection system based on code similarity analysis. In: Proceedings of the 32nd annual conference on computer security applications. pp. 201–213 (2016)
16. Na, Y., Woo, S., Lee, J., Lee, H.: CNEPS: A Precise Approach for Examining Dependencies Among Third-Party C/C++ Open-Source Components. In: Proceedings of the 46th International Conference on Software Engineering (ICSE). pp. 2918–2929 (2024)
17. Servant, F., Jones, J.A.: Fuzzy Fine-Grained Code-History Analysis. In: Proceedings of the 39th International Conference on Software Engineering (ICSE). pp. 746–757 (2017)

18. Shi, Y., Zhang, Y., Luo, T., Mao, X., Yang, M.: Precise (Un) Affected Version Analysis for Web Vulnerabilities. In: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering. pp. 1–13 (2022)
19. Sun, Q., Xu, L., Xiao, Y., Li, F., Su, H., Liu, Y., Huang, H., Huo, W.: VERJava: Vulnerable Version Identification for Java OSS with a Two-Stage Analysis. In: 2022 IEEE International Conference on Software Maintenance and Evolution (ICSME). pp. 329–339 (2022). https://doi.org/10.1109/ICSME55016.2022.00037
20. Tang, L., Ni, C., Huang, Q., Bao, L.: Enhancing bug-inducing commit identification: A fine-grained semantic analysis approach. IEEE Transactions on Software Engineering **50**(11), 3037–3052 (2024). https://doi.org/10.1109/TSE.2024.3468296
21. Woo, S., Choi, E., Lee, H.: A Large-Scale Analysis of the Effectiveness of Publicly Reported Security Patches. Computers & Security **148**, 104181 (2025)
22. Woo, S., Choi, E., Lee, H., Oh, H.: V1SCAN: Discovering 1-day Vulnerabilities in Reused C/C++ Open-source Software Components Using Code Classification Techniques. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 6541–6556 (2023)
23. Woo, S., Hong, H., Choi, E., Lee, H.: MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 3037–3053 (2022)
24. Woo, S., Lee, D., Park, S., Lee, H., Dietrich, S.: V0Finder: Discovering the Correct Origin of Publicly Reported Software Vulnerabilities. In: 30th USENIX Security Symposium (USENIX Security 21). pp. 3041–3058. USENIX Association (Aug 2021)
25. Woo, S., Park, S., Kim, S., Lee, H., Oh, H.: CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse. In: Proceedings of the IEEE/ACM 43rd International Conference on Software Engineering (ICSE). pp. 860–872 (2021)
26. Wu, S., Wang, R., Huang, K., Cao, Y., Song, W., Zhou, Z., Huang, Y., Chen, B., Peng, X.: Vision: Identifying Affected Library Versions for Open Source Software Vulnerabilities. In: Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering. pp. 1447–1459 (2024)
27. Xiao, Y., Chen, B., Yu, C., Xu, Z., Yuan, Z., Li, F., Liu, B., Liu, Y., Huo, W., Zou, W., Shi, W.: MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures. In: Proceedings of the 29th USENIX Security Symposium (Security). pp. 1165–1182 (2020)
28. Yamaguchi, F., Golde, N., Arp, D., Rieck, K.: Modeling and Discovering Vulnerabilities with Code Property Graphs. In: Proceedings of the 35th IEEE Symposium on Security and Privacy (SP). pp. 590–604. IEEE (2014)
29. Zhan, X., Liu, T., Fan, L., Li, L., Chen, S., Luo, X., Liu, Y.: Research on Third-Party Libraries in Android Apps: A Taxonomy and Systematic Literature Review. IEEE Transactions on Software Engineering **48**(10), 4181–4213 (2022). https://doi.org/10.1109/TSE.2021.3114381
30. Zou, D., Wang, S., Xu, S., Li, Z., Jin, H.: VulDeePecker: A Deep Learning-Based System for Multiclass Vulnerability Detection. IEEE Transactions on Dependable and Secure Computing **18**(5), 2224–2236 (2021). https://doi.org/10.1109/TDSC.2019.2942930