2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)

TIVER: Identifying Adaptive Versions of C/C++ Third-Party Open-Source Components Using a Code Clustering Technique

Youngjae Choi
Korea University
Republic of Korea
youngjaechoi@korea.ac.kr

Seunghoon Woo*
Korea University
Republic of Korea
seunghoonwoo@korea.ac.kr

***Abstract*—Reusing open-source software (OSS) provides significant benefits but also poses risks from propagated vulnerabilities. While tracking OSS component versions helps mitigate threats, existing approaches typically map a single version to the reused codebase. This coarse-grained approach overlooks the coexistence of multiple versions, leading to ineffective OSS management. Moreover, identifying component versions is further complicated by noise codes, such as shared algorithmic code across different OSS, and duplicate components caused by redundant OSS reuse.**

In this paper, we introduce the concept of the adaptive version, a one-stop solution to represent the version diversity of reused OSS. To identify adaptive versions, we present TIVER, which employs two key techniques: (1) fine-grained function-level versioning and (2) OSS code clustering to identify duplicate components and remove noise. This enables precise identification of OSS reuse locations and adaptive versions, effectively mitigating risks associated with OSS reuse. Evaluation of 2,025 popular C/C++ software revealed that 67% of OSS components contained multiple versions, averaging over three versions per component. Nonetheless, TIVER effectively identified adaptive versions with 88.46% precision and 91.63% recall in duplicate component distinction, and 86% precision and 86.84% recall in eliminating noise, while existing approaches barely achieved 42% recall in distinguishing duplicates and did not address noise. Further experiments showed that TIVER could enhance vulnerability management and be applied to Software Bills of Materials (SBOM) to improve supply chain security.

***Index Terms*—Open-Source Software, Third-Party Library Management, Version Identification, Supply Chain Security**

I. INTRODUCTION

Software reuse has become a cornerstone of modern software development, with open-source software (OSS) accelerates development and reduces costs [1]–[4]. However, improper OSS reuse can compromise the entire system (*e.g.*, vulnerability propagation [5]–[10]). A key strategy to mitigate this is identifying and tracking reused OSS versions and updating vulnerable components (*e.g.*, [11]–[14]).

In a real-world software ecosystem, especially in C/C++ software where code-level reuse is more dominant than package-level reuse, OSS is rarely reused without modifications [4], [15]. Developers often use only parts of OSS, apply backported patches, or unintentionally include redundant copies due to nested dependencies [4], [16]. This *adaptive* reuse results in a complex landscape where multiple OSS versions coexist within a program, complicating version identification and security management.

Therefore, identifying the "***adaptive version***" of OSS components — a comprehensive representation that encompasses the various versions present in reused code — is crucial for software security. However, this task is challenging by noise and duplicate components, which can result in inaccurate version identification and flawed vulnerability assessments.

- **Noise.** Code snippets that are commonly present in various OSS due to their brevity or the implementation of widely used algorithms.
- **Duplicate components.** Cases where the same OSS is reused in multiple parts of the target software, often with different versions to satisfy specific requirements [16].

Removing noise ensures accurate version identification by preventing unrelated code from being misclassified as OSS [17]–[19], reducing false alarms in vulnerability reports. Detecting duplicate components allows precise mapping of OSS usage, each with unique vulnerabilities. Failure to distinguish duplicates can lead to overlooked version-specific issues and compromised vulnerability assessments.

Limitations of existing approaches. To the best of our knowledge, no existing approaches have considered the version diversity of C/C++ components while attempting to distinguish duplicate components and eliminate noise (see Section VI). For example, CENTRIS [4] and V1SCAN [12] map only the most prevalent version within a component, without considering duplicates and noise. CNEPS [16] made some progress by partially identifying duplicate components; however, it did not perform version identification and gave little consideration to noise. While OSSFP [20] attempted to eliminate noise from the perspective of component identification, it failed to apply this to version prediction and did not consider version diversity and duplicate components. This gap in current research underscores the need for a more comprehensive solution to identify C/C++ OSS component versions effectively.

* Corresponding author

1558-1225/25/$31.00 ©2025 IEEE
DOI 10.1109/ICSE55347.2025.00188

Our approach. We present TIVER (adapTIve version analyzER), a novel approach for the comprehensive identification of adaptive versions in C/C++ OSS components.

The key concepts of TIVER, which are markedly distinct from existing approaches, are (1) fine-grained versioning at the function level and (2) an OSS code clustering technique.

Given a target software, TIVER identifies the OSS components and specifies the OSS version for each reused function (see Section III-B). It adapts CENTRIS [4] to identify components and then compares the codebase of the target software with that of each OSS version. This process enables TIVER to determine the specific versions associated with reused functions. TIVER then utilizes OSS code clustering to isolate the reused OSS code regions (see Section III-D). It groups code segments expected to belong to the same OSS, based on the file names and directory structures in which the reused functions are located, thereby creating clusters. Subsequently, clusters with duplicate source files are identified as duplicate components, whereas clusters with a small proportion of reused functions are considered noise and pruned. Finally, TIVER aggregates the versions of the reused functions within each cluster and identifies the adaptive version that encompasses OSS version diversity (Section III-E).

Evaluation. When we applied TIVER to 2,025 C/C++ software projects on GitHub, we observed that OSS components with a single version accounted for only 33% of the total. Instead, each component had more than *three* distinct versions, and 12% of the identified components were reused redundantly.

Despite various OSS reuse patterns in which different versions coexist and redundant reuse occurs, TIVER can effectively identify adaptive versions encompassing multiple coexisting OSS versions. In particular, TIVER showed 88.46% precision and 91.63% recall in duplicate component distinction and 86% precision and 86.84% recall in noise elimination, whereas existing approaches barely achieved 42% recall in duplicate component distinction and failed to address noise (see Section IV-A). We further demonstrate that by integrating the adaptive versions identified by TIVER with the Software Bill of Materials (SBOM) [21], TIVER can be used to improve supply chain security, especially in detecting propagated vulnerabilities (see Section IV-D).

Contributions. We summarize our contributions below.

- We present TIVER, the first approach to effectively identify adaptive versions of reused OSS components, using fine-grained versioning and OSS code clustering to distinguish duplicate components and eliminate noise.
- TIVER revealed that OSS components with a single version made up only 33% of our dataset, emphasizing the need to address multiple versions and proposing a detailed process for identifying adaptive versions.
- Experiments on popular software show that TIVER effectively identifies adaptive versions, distinguishes duplicate components, and eliminates noise. We demonstrated that TIVER can enhance supply chain security and vulnerability management when used with the SBOM.

II. MOTIVATION

A. Terminology

We define several terms upfront.

- **OSS reuse.** This refers to utilizing a portion of OSS functions or the entire OSS source code [4], [22].
- **OSS component.** An OSS component is a set of OSS functions reused in a target program [4].
- **OSS version.** We define an OSS version to adhere to the default three-component semantic versioning notation of `major.minor.patch` [23].
- **OSS update.** We consider any change in the aforementioned three-component (*i.e.*, `major.minor.patch`) of an OSS version to constitute an update of the OSS.

B. Problem and goal statements

Problem. In this paper, we aim to address the problems that occur when mapping a single version to a C/C++ OSS component. In C/C++ languages, it is typical to reuse OSS with code modifications [4], [5], [8], which involve updating specific portions of the code to newer versions (*e.g.*, through backporting patches). Consequently, multiple versions of code may exist within a single OSS component.

However, OSS management becomes *inefficient* when determining the version of OSS components as a single entity. One of the critical issues is the difficulty in precisely identifying vulnerabilities. As identified by V1SCAN [12], mapping a single version to the entire component without considering version diversity results in a 77% false positive rate for vulnerability detection. Moreover, this can overlook certain vulnerabilities and potentially leave unidentified threats.

Goal. Therefore, we aim to identify the issues arising when mapping a single version to OSS components and propose a solution to identify *adaptive versions* of C/C++ OSS components for efficient third-party library management. Additionally, we intend to compile the issues that arise during the identification of C/C++ OSS component versions and propose policies that encompass diverse versions of reused OSS codes.

C. Motivating example

To demonstrate the importance of identifying an adaptive version, we attempted to manage the OSS components of `ReactOS`[1], a free Windows-compatible operating system.

As of March 2024, the master version of `ReactOS` reused 68 source files from `Libxml2`[2]. When the version of each reused source file was identified by referring to the commit history, six versions emerged. The distribution of the reused versions is listed in Table I.

Limitations of existing approaches. Existing approaches [4], [12] designed to identify the versions of reused OSS components (at the source code level) have primarily focused on the version to which the majority of functions belong. However, this method results in inefficient vulnerability discovery.

[1] https://github.com/reactos/reactos
[2] https://gitlab.gnome.org/GNOME/libxml2

TABLE I: Version distribution of reused `Libxml2` source files in `ReactOS` (as of March 2024).

Version	#Reused files	Ratio
v2.9.10	4	6%
v2.9.12	7	10%
v2.10.0	**48**	**71%**
v2.10.1	1	1%
v2.10.2	2	3%
v2.10.3	6	9%
Total	68	100%

For example, when CENTRIS [4] was used, the reused version of `Libxml2` was v2.10.0. Applying this result to vulnerability detection, it can be concluded that `ReactOS` contains two propagated vulnerabilities, CVE-2022-40303 and CVE-2022-40304, which were reported to be present in `Libxml2` versions prior to v2.10.2. However, the reused files containing these vulnerabilities were updated by the `ReactOS` team to v2.10.3 by backporting the security patches. Hence, these results are false positives (FPs) because the vulnerabilities are remediated. Similarly, if vulnerabilities are contained within source files belonging to versions v2.9.12 or v2.9.10, existing approaches overlook these vulnerabilities.

TIVER. TIVER can distinguish multiple versions mixed in reused OSS codes and identify adaptive versions of OSS components, thereby enabling effective vulnerability responses. TIVER uses code clustering to (1) precisely identify reused OSS code areas and (2) eliminate noise that hinders version identification. For example, TIVER detects that the source files under the `ReactOS/dll/3rdparty/libxslt` path contain `Libxml2` code because both libraries share a commonly used algorithmic code. TIVER identifies these as noise and excludes them from the `Libxml2` version identification. Furthermore, if `Libxml2` is redundantly reused in `ReactOS`, TIVER determines the adaptive version for each reused code area (see Section III-D). Finally, TIVER identifies the adaptive version of `Libxml2` (*i.e.*, ^2.9.10; see Section III-E) and tracks which version the reused functions belong to. This ensures accurate component version tracking, even when specific functions are updated or experience delayed updates, thus aiding effective library management. For instance, TIVER can identify that the reused functions containing the two aforementioned CVEs are from a version where vulnerabilities have been patched, thereby preventing FPs. Moreover, it can detect vulnerabilities present in versions prior to v2.10.0 (an in-depth analysis of these issues is presented in Section IV-D).

III. DESIGN OF TIVER

We describe the design of TIVER, an effective approach for identifying adaptive versions of C/C++ OSS components.

Considerations. Before introducing TIVER in detail, we present two important considerations that need to be considered for the effective management of OSS by identifying adaptive versions.

First, we address noise, which refers to common source code found in various OSS projects. Since code that implements common algorithms (*e.g.*, cryptographic functions) appears frequently in various OSS projects, this noise must be filtered out to prevent misidentification. Our approach carefully separates such common code from the unique characteristics that distinguish specific OSS versions.

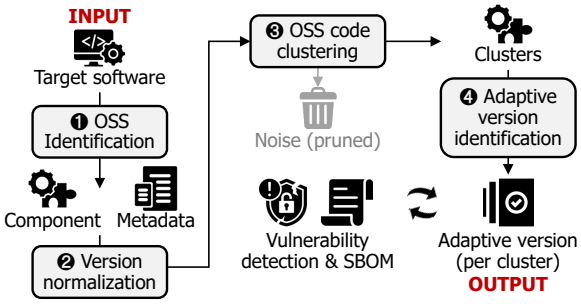


Fig. 1: High-level overview of TIVER.

Second, it is important to distinguish duplicate components. The target software may reuse the same OSS but different versions in various code sections to meet specific requirements. In these cases, it is crucial to distinguish duplicate components and carefully verify the version of each code section for efficient OSS management (*e.g.*, vulnerability assessment).

A. Approach overview

Figure 1 shows the high-level workflow of TIVER. The distinguishing features of TIVER from existing approaches include (1) considering fine-grained versioning at the function level and (2) eliminating noise and distinguishing duplicate components through *OSS code clustering*.

TIVER begins by identifying the codebase of each reused OSS within the target software, accomplished by utilizing CENTRIS [4], an OSS component identification tool.

Once the reused OSS components are identified, TIVER assigns versions based on the granularity of individual functions rather than treating the entire component as a single entity. This fine-grained versioning enables TIVER to capture the diversity of versions that may exist in a single component.

After assigning a version to each reused function, TIVER employs *OSS code clustering* to distinguish *noise* and *duplicate components* from the reused OSS code areas.

Finally, TIVER identifies an adaptive version that encompasses diverse versions of the reused OSS code areas. This is achieved by analyzing the versions of individual functions within each component (excluding noise) and establishing a representative version (or range) for the entire component. In cases where duplicate components exist, adaptive versioning is performed for each component.

Design assumption. TIVER identifies adaptive versions at the source code level. TIVER can be applied to any granularity level (*e.g.*, files and functions); however, we focused on *function* units. Function granularity is ideal for identifying OSS components and adaptive versions as it reduces false negatives (FNs) in OSS and version identification compared to coarser units (*e.g.*, files) and minimizes false positives compared to finer units (*e.g.*, lines) [4], [24].

B. OSS identification

The initial phase involved identifying OSS components within the target software. Because our goal is to identify an adaptive version rather than precisely discover OSS components, we leveraged CENTRIS [4] due to its ability to identify modified OSS components, as well as its publicly accessible source code and dataset [25].

However, the original CENTRIS tool provided only a list of component names and reused source files. To align with our goal, we modified the CENTRIS source code to include the information listed in Table II in the output.

TABLE II: Modified output of CENTRIS.

Element	Description
Component name	The name of the identified OSS component.
Common functions	Functions commonly present between each identified component and the target software.
Path information	The directory path of common functions in the target program.
Segmented OSS versions	Versions to which each common (reused) function belongs in the original OSS.

The CENTRIS dataset lists the versions to which OSS functions belong, making it easy to extract segmented OSS versions. Note that CENTRIS focuses on component identification and does not engage in adaptive version identification.

C. Version normalization

This paper considers three-component semantic versioning (*i.e.*, `major.minor.patch`; see Section II-A). However, versions can be handled in various ways in an OSS ecosystem. Hence, TIVER normalizes versions to standardize heterogeneous version strings into a consistent format.

The algorithm operates in the following three steps.

S1. Numeric element extraction. TIVER first isolates numeric elements from the input string. TIVER disregards non-numeric characters that may vary across different versioning schemes (*e.g.*, in the case of "`v0.0.1`," the non-numeric character "`v`" is ignored).

S2. Semantic version construction. TIVER constructs a standardized semantic version. The first extracted number is assigned as `major`. If `major` is present, the second number is assigned to the `minor` version. TIVER then parameterizes the delimiter of numbers (*e.g.*, "." or "_"). The third (or last) number is assigned as the `patch` version and checked for delimiter consistency.

S3. Validation and augmentation. TIVER validates and augments the constructed semantic version. When the `major` and `minor` versions are present but `patch` is missing, TIVER assigns "0" as the `patch` (*i.e.*, zero-padding). After applying the above steps, the version is flagged as invalid if any element is missing.

For instance, `OpenSSL_1_1` was normalized to a semantic version with a `major` of 1, `minor` of 1, and `patch` of 0. This normalization enhances the ability to standardize version information across various OSS projects, thereby improving the effectiveness of subsequent analyses.

D. OSS code clustering

To efficiently identify adaptive versions, TIVER employs an OSS code *clustering* technique that aims to identify noise and distinguish duplicate components. To achieve this, TIVER leverages the following two key intuitions.

(1) Noisy regions contain only a few reused functions.

(2) Duplicate OSS codebases contain redundant files.

We use Google's `Filament` (commit ID `ce7dd7`) as a working example (see Figure 2). We assume a situation in which noise and duplicate components are identified through clustering for *one* component, and the working example introduces the process of clustering for `GoogleTest` reused in `Filament`. The clustering consisted of four steps.

Step 1: Directory hierarchy identification. The initial step is to identify the directory structure of the target software and determine where it reuses functions from the OSS project, using the paths of the reused functions recorded in advance (see Section III-B). The results are shown in a *tree* structure, where the root node is the target software, the leaf nodes are the reused source files from the OSS, and the inner nodes are the target software directories. For example, Figure 2a shows a part of the tree for `Filament`. Because `geometry` uses general testing logic similar to `GoogleTest`, Figure 2a indicates that the `geometry` directory also includes functions reused from `GoogleTest` (*i.e.*, noise).

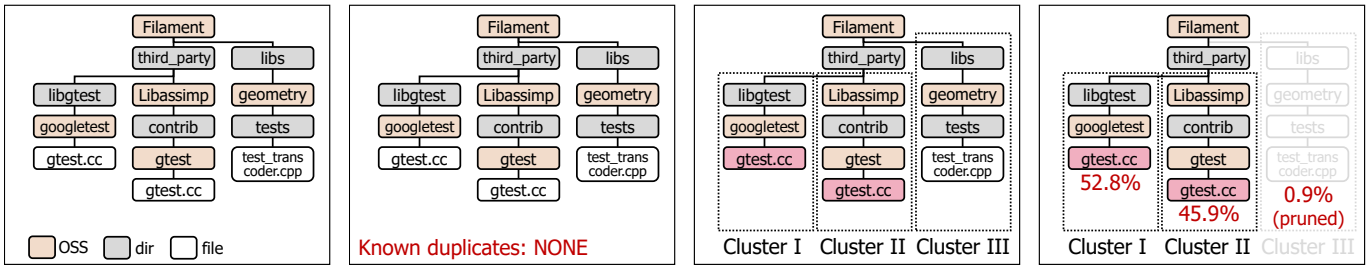
Step 2: Known duplicates examination. To cluster the reused code, TIVER utilizes the names of reused files: if OSS is redundantly reused, files with the same name may coexist in the target software (this is discussed in Section V-B). However, duplicate files may exist in the original OSS. TIVER defines duplicated files present in the original OSS code before reuse as *known duplicates*. To effectively distinguish duplicate components in later steps, TIVER identifies known duplicates and their occurrences for each OSS using the CENTRIS database (see Section IV). If the occurrences of duplicates varied across OSS versions, TIVER considered the maximum value.

Step 3: Reused code clustering. Reused code clustering was performed based on three main rules.

R1. By default, all child nodes that share the same Level-1 parent node are assigned to the same cluster.

R2. If files included in *known duplicates* redundantly exist in the tree and their occurrences exceed the number specified in *known duplicates*, TIVER locates the Lowest Common Ancestor (LCA) and uses the LCA node as a reference node to assign different clusters at the immediate subordinate directory level. This operation is performed only on directories containing duplicate files.

R3. If files that are not listed in *known duplicates* exist redundantly in the tree, regardless of their occurrences, TIVER locates the LCA and uses the LCA node as a reference node to assign different clusters at the immediate subordinate directory level. This operation is performed only on directories containing duplicate files.



(a) Directory hierarchy identification. **(b) Known duplicates (of `GoogleTest`) examination.** **(c) Reused code clustering based on *unknown* duplicates.** **(d) Cluster pruning based on code occupancy.**

Fig. 2: Diagrams of OSS code clustering. These sequentially depict the clustering for the `GoogleTest` reused by `Filament`, and the goal is to identify noise and distinguish duplicate components. Note that only some nodes are shown in each tree for ease of explanation (*e.g.*, in Figure 2d, because some clusters are omitted, the sum of clusters' total proportions is less than 100%).

TIVER essentially performs clustering from lower to higher levels, down to a level excluding the root node of the tree. This involves starting each cluster from the Level-1 directory to identify the noise that generally exists in paths different from the OSS code area more efficiently.

Next, TIVER focuses on duplicate files. For the *known duplicates*, clustering was performed by comparing the number of occurrences. In contrast, the presence of *unknown* duplicates indicates that an OSS is redundantly reused; thus, clustering is performed regardless of its occurrence. We demonstrated that filename-based clustering works effectively (see Section IV-A), and further issues are discussed in Section V-B.

In our working example, all descendant directories and files of `third_party` and `libs` are inherently grouped into their respective clusters. TIVER verified that the original `GoogleTest` contains no known duplicates. However, there are *unknown* duplicates (*i.e.*, `gtest.cc`) in the `third_party` cluster; thus, TIVER locates the LCA (*i.e.*, `third_party`) and creates two separate clusters (*i.e.*, `third_party/libgtest/` and `third_party/Libassimp/`; see Figure 2c). Because the rightmost cluster does not contain any files common to the other clusters, according to **R1**, it is clustered under `libs`. Using this method, TIVER can identify the boundaries of OSS component codebases within the target software.

Step 4: Cluster pruning. After clustering the tree, TIVER performs pruning to eliminate noise. To achieve this, TIVER measures the proportion of the reused functions in each cluster. TIVER considers the total number of functions reused in the target software to be 100%. TIVER calculates their respective proportions based on the number of functions included in each cluster. If the proportion of functions in any cluster falls below a threshold θ (typically a small value), TIVER considers this cluster noise and removes it from the directory tree (experiments on the threshold sensitivity are presented in Section IV-B). For example, in Figure 2d, if we set θ to 0.03 (3%), `Cluster III` is pruned because the proportion of reused functions is 0.3% (*i.e.*, less than 3%). After this process, the remaining clusters are identified as reused OSS code areas. If two or more clusters remain, TIVER decides that it is a duplicate component and obtains an adaptive version for each cluster area.

E. Adaptive version identification

Finally, TIVER assessed the adaptive versions of each cluster. TIVER operates on the principle of identifying the most conservative version range, which encompasses all versions present within a given cluster.

Let $V = \{v_1, v_2, ..., v_n\}$ be the set of all unique versions (v_i) of the functions included in a cluster. When a function belongs to multiple versions with the same syntax, only the latest version is considered (this is discussed in Section V-A). Version v_i can be represented as a 3-tuple: (m_i, n_i, p_i), where m_i, n_i, and p_i represent the `major`, `minor`, and `patch`, respectively. Let v_o be the oldest version of V, and let v_x be an invalid version (*i.e.*, a version that failed to normalize). The adaptive version v_a of the cluster is obtained as follows.

$$v_a = \begin{cases} v_o & \text{if } V = \{v_o\} \\ {}^{+}v_o & \text{if } V = \{v_o, v_x\} \\ {}^{*}v_o & \text{if } \exists v_i, v_j \in V.(m_i \neq m_j) \\ {}^{\wedge}v_o & \text{if } (\forall v_i, v_j \in V.(m_i = m_j)) \wedge (\exists v_i, v_j \in V.(n_i \neq n_j)) \\ {}^{\sim}v_o & \text{if } (\forall v_i, v_j \in V.((m_i = m_j) \wedge (n_i = n_j))) \\ & \quad \wedge (\exists v_i, v_j \in V.(p_i \neq p_j)) \end{cases}$$

TIVER expands upon the semantic versioning of the Node Package Manager (`npm` [26]) by incorporating additional elements for more efficient version notation.

First, if all functions in a cluster share the same version, which is not an invalid version, TIVER determines that version to be the cluster version. Next, if there are one or more invalid versions present in the cluster's version set, but excluding the invalid versions, if the cluster has only one unique version (v_o), TIVER denotes the version of the cluster as ${}^{+}v_o$.

When there is more than one normalized version in the cluster, TIVER uses the following three notations: $*$, $\hat{}$, and $\sim$. These notations are similar to the ones used in `npm`, but we clarify the definition of each notation while simplifying it further to avoid confusion with the existing conventions. The scope of the three notations defined by TIVER is as follows.

- "*" allows for any `major`, `minor`, or `patch` increments.
- "^" allows for `minor` and `patch` increments.
- "~" allows only `patch` increments.

For example, consider an OSS component X (*i.e.*, cluster) with the following versions: [1.2.0, 1.2.5, 1.3.2, 1.4.0]. The adaptive version for this component would be ^1.2.0, allowing

`minor` and `patch` increments from 1.2.0 (*i.e.*, v_o) onwards, such as 1.2.5, and 1.4.0, but not 2.0.0 or higher.

As part of supply chain security through using SBOM [21], TIVER manages the version of all functions contained within each cluster in a separate file (an example is presented in Listing 4). While the adaptive version can provide a rough idea of which versions of source files have been reused, this information allows developers to address propagated vulnerabilities more easily.

For example, suppose a CVE vulnerability exists in version 1.4.1 of the abovementioned OSS component X. Because the adaptive version is identified as ^1.2.0, developers should determine whether this vulnerability has been propagated. In this case, TIVER's function-level versioning can be leveraged. If the version of the reused function (associated with the CVE) is not 1.4.1, developers can confirm that the vulnerability has not been propagated. Conversely, if the function's version is 1.4.1, developers can take appropriate actions, such as applying security patches. The practicality of TIVER in vulnerability verification and management is introduced in Section IV-D.

IV. EVALUATION

In this section, we evaluate TIVER based on the following four research questions.

- **RQ1: Accuracy.** How precise are the algorithms used by TIVER for adaptive version identification?
- **RQ2: Effectiveness.** How effective is TIVER in identifying an adaptive version that encompasses all the versions present within each cluster?
- **RQ3: Performance.** How does TIVER perform in terms of runtime and resource usage when analyzing large-scale OSS components with complex version usage patterns?
- **RQ4: Practicality.** How can TIVER be utilized for supply chain security (*e.g.*, SBOM) and vulnerability detection?

We ran TIVER on a machine with a GNU/Linux 6.5.0-41-generic x86_64, Intel (R) Core (TM) i9-14900K @ 5.70GHz, 64GB RAM, and a 2TB SSD.

Architecture of TIVER. TIVER consists of two modules: a *target parser* and an *adaptive version analyzer*. The *target parser* parses the C/C++ source code in the target software, extracts, and hashes every function contained (for applying CENTRIS). It utilizes Ctags [27], a robust regular expression-based parser, to identify and extract the functions. The *adaptive version analyzer* identifies adaptive versions using reused code clustering. It utilizes the `Anytree` library [28], which provides efficient data structures and algorithms for building and manipulating tree structures. TIVER was implemented in Python, with its main functionality spanning approximately 1,000 lines of code, excluding external libraries.

OSS dataset. To operate TIVER, CENTRIS must first identify OSS components. Hence, we leveraged a dataset containing 10,417 OSS projects [25], originally curated in April 2020 by CENTRIS and updated in April 2022 by V1SCAN. This dataset includes functions present in the OSS projects across all their versions. Because our goal is not limited to identifying adaptive versions in only the latest systems, we determined that conducting experiments with this dataset does not raise critical concerns regarding the evaluation of TIVER.

Target software selection. To demonstrate the generality of TIVER, we selected popular software from GitHub as our target. We collected the **top 2,025 GitHub C/C++ software packages** based on their stargazer counts (*i.e.*, a popularity indicator). We determined that this large-scale target dataset, with a total of 570 million lines of code, was suitable for evaluating the effectiveness, accuracy, and performance of TIVER. To effectively utilize the OSS dataset, we evaluated the target software using an older version (*i.e.*, the version released closest to April 2022 for each target software).

A. Accuracy of TIVER

Methodology. Because we introduce the concept of *adaptive versions* for the C/C++ components for the first time, there is no existing ground truth available. Moreover, defining criteria for accuracy assessment is complex due to the fact that any reused function's version falls within the adaptive version, while functions from versions outside this range are not reused.

Therefore, we evaluated the accuracy of the following key techniques crucial for TIVER: (1) duplicate component distinction, (2) noise elimination, and (3) version normalization. Because there is no ground truth available, we manually examined all results with two analysts: one with over 10 years of experience in software engineering and security, and the other with over three years. Initially, we verified CENTRIS's accuracy by cross-referencing the OSS and directory names of the reused functions. Next, we measured the FPs and FNs of TIVER. We can closely examine TIVER's results because TIVER provides all the reused components' functions and files in a tree structure. Specifically, we reviewed the paths and names of the directories and files in the clusters, referring to the source code when necessary. If a non-duplicated component is split during clustering or if non-noise code is eliminated, it is considered an FP of TIVER. Conversely, if undistinguished duplicate components or unremoved noise are observed, it is classified as an FN of TIVER. Any discrepancies between the analysts were resolved through discussion. We implemented TIVER on the target software using a pruning threshold θ of 0.03 (see Section IV-B for details).

Adaptive version identification results. Among the 3,109 components identified in the target software by CENTRIS, 1,191 (38.3%) OSS results were FPs and were therefore excluded. Note that the results of CENTRIS are independent of the performance of TIVER. Even if CENTRIS identifies incorrect components, TIVER can treat the misidentified function set as a component and precisely identify adaptive versions. However, because a misidentified function set is not particularly meaningful in practical applications, we decided to exclude these results.

For the remaining 1,918 OSS components, TIVER identified 3,351 clusters and found that 12% (230 of 1,918) of the

TABLE III: Accuracy of the TIVER's algorithms for 2,025 target software. TIVER identified adaptive versions of 3,351 clusters (after pruning) among 1,918 reused OSS components by effectively identifying duplicate components and removing noise.

Tool	#Target software		#Detected components	#Total clusters	Duplicate component distinction					Noise elimination				
	Total	Including components			#TP	#FP	#FN	Precision	Recall	#TP	#FP	#FN	Precision	Recall
TIVER	2,025	869	1,918	3,351	230	30	21	88.46%	91.63%	264	43	40	86.00%	86.84%

Listing 1: Example of an FP in duplicate component distinction. Note that the original `pybind11` contains only ".cpp" files, but the `TNN` team created ".cc" files by cloning the original files.

```
TARGET: TNN (https://github.com/Tencent/TNN)
OSS: pybind11 (https://github.com/pybind/pybind11)
===
TNN/tools/onnx2tnn/onnx-converter/pybind11/test/
 ⊢ test_buffers.cc (file)
 ⊢ test_buffers.cpp (file)
 ⊢ test_class.cc (file)
 ⊢ test_class.cpp (file) ...
```

identified components were redundantly reused. In addition, TIVER identified 264 noise clusters, removed them, and identified the adaptive versions of the remaining clusters. Table III summarizes the accuracy of the TIVER algorithm.

Duplicate component distinction accuracy. TIVER identified 273 duplicate components out of 1,918 reused components. Among these, 230 were true positives (TPs), demonstrating a precision of 88.46% and recall of 91.63%.

Most FPs occurred when developers cloned a reused code and maintained it under the same file name but with a different extension (see Listing 1). TIVER misinterprets these instances as duplicate components and separates them into clusters. Although these could be considered duplicates, they were not cases where developers reused OSS in multiple locations. Therefore, we classified them as FPs for TIVER.

In contrast, FNs occurred when the OSS was reused disjointly (except for the same file) or when the file name was completely changed while being reused. Although this was not common, and thus FNs were relatively lower than FPs, in these cases, TIVER failed to correctly separate the clusters, resulting in FNs in duplicate component identification.

We further compared TIVER with CNEPS [16]. While CNEPS focuses on dependencies rather than versions, it identifies duplicates based on function-call relationships. If a function in an OSS is called from multiple locations with different header paths, it is considered a duplicate component. We evaluated ten target software: five with three or more TPs and five with at least one FN in the TIVER results. In CNEPS, the FP criterion is ambiguous because detecting multiple dependencies does not necessarily imply that they are duplicates, our focus was on comparing TPs and FNs.

Table IV summarizes the comparison results. Notably, CNEPS did not detect any duplicates that TIVER missed. TIVER precisely distinguished most of the duplicate components, showing higher recall than CNEPS. In contrast, CNEPS produced many FNs, such as when call relationships were ambiguous or when duplicate components existed but the callee functions were different, resulting in only 42% recall.

TABLE IV: Accuracy comparison between TIVER and CNEPS.

Target software	TIVER			CNEPS [16]		
	#TP	#FN	#Recall	#TP	#FN	#Recall
OpenBSD	9	0	1.00	3	6	0.33
Node-packer	8	0	1.00	5	3	0.63
AliOS-Things	6	0	1.00	1	5	0.17
FreeBSD	5	1	0.83	1	5	0.17
Overgrowth	5	0	1.00	4	1	0.80
YDB	3	1	0.75	2	2	0.50
MAME	2	1	0.67	2	1	0.67
Fastsocket	2	1	0.67	1	2	0.33
Blender	1	1	0.50	1	1	0.50
Urho3D	1	1	0.50	0	2	0.00
Total	**42**	**6**	**0.88**	20	28	0.42

Noise elimination. Next, we evaluated the accuracy of the noise elimination process. TIVER identified 307 clusters as noise from all detection results. Among these, 264 were TPs, resulting in a precision of 86% and recall of 86.84%.

Most FPs in noise elimination were caused by reasons similar to those in distinguishing duplicate components. When developers reuse OSS code and clone it under the same file name, TIVER locates the cloned code as separate clusters. If the proportion of functions included in this cluster is less than θ, TIVER considers them noise (see Section III-D), resulting in FPs in noise elimination.

Establishing clear criteria for identifying FNs is challenging. We considered cases where code was not explicitly contained in OSS (*e.g.*, referring to its repository) but was not pruned in noise elimination as FNs. Most FNs occurred when the noise cluster shared the same Level-1 node with the reused OSS code regions but had different file names from the reused OSS codebases. In addition, they occurred in cases where the code was not explicitly part of the OSS code region but shared sufficient code to avoid pruning (*e.g.*, generating test files by referencing the testing logic of `GoogleTest`). In such cases, TIVER fails to eliminate noise, thereby yielding FNs.

Although TIVER yielded false alarms in certain scenarios, to our knowledge, no attempts have been made to eliminate noise in OSS version identification in the presence of duplicate components. We believe that TIVER is effective because it can reduce most of the noise through code clustering.

Version normalization. We examined the results of version normalization using 4,720,744 version strings, which represent the versions of all the reused functions in the detected OSS components. First, TIVER reported that 61.64% (2,890,772) of its versions aligned with the normalization algorithm without any issues. Additionally, 26.35% (1,235,802) of the versions were successfully normalized through zero-padding. The remaining 12% of the version strings were deemed invalid.

TABLE V: Effectiveness of TIVER.

Approaches	#Target	#Components	VDI	NCI
V1SCAN [12] (baseline)	869	1,918	1	0
TIVER			**3.49**	**3.31**

Most of the invalid versions consisted solely of strings, such as tags created for testing purposes (*e.g.*, "`ms-bug-test`"). We assert that identifying invalid versions is not a shortcoming of TIVER but rather a necessary step to exclude versions unrelated to semantic versions when identifying adaptive versions. Therefore, this underscores the effectiveness of the TIVER version normalization algorithm.

Answer to RQ1. TIVER effectively identified the adaptive versions of reused OSS code locations by (1) distinguishing duplicate components with 88.46% precision and 91.63% recall, (2) pruning noise with 86% precision and 86.84% recall, and (3) effectively distinguishing between valid and invalid versions through version normalization.

B. *Effectiveness of* TIVER

Methodology. To demonstrate the effectiveness of TIVER, we compared it with V1SCAN [12], which detects vulnerabilities based on the prevalent version of detected components. Since V1SCAN only maps one version to a component, instead of conducting an unfair experiment criticizing V1SCAN, we assessed TIVER by comparing it to a baseline that maps a single version without considering duplication and noise.

In this context, we consider the following two metrics.

1. **Version diversity index (VDI).** This metric indicates how well an approach can cover version diversity. It represents the extent to which different versions coexist.

$$\texttt{VDI} = \frac{\textit{\#Identified distinct OSS versions}}{\textit{\#OSS components}}$$

2. **Noise cleansing index (NCI).** This index indicates how well an approach addresses unnecessary versions in the noise that hinders precise version identification.

$$\texttt{NCI} = \frac{\textit{\#Unique OSS versions contained in the noise}}{\textit{\#OSS components}}$$

Result analysis. Table V summarizes the measurement results. Unlike V1SCAN that mapped only one version to a component (VDI: 1), TIVER confirmed that, on average, more than three versions (VDI: 3.49) coexist and proposed adaptive versions to encompass them. Compared to V1SCAN, which does not consider duplicates and noise (NCI: 0), TIVER distinguishes duplicate components and identifies the versions of the reused OSS by removing more than three noisy versions (NCI: 3.31). In summary, TIVER covers more versions than the baseline and removes noise that interferes with version identification, thereby demonstrating its effectiveness in OSS management.

Case study. We demonstrate TIVER's effectiveness with a case study of `Unbound` OSS in `OpenBSD` (commit ID `555665`). `Unbound` was redundantly used in two paths: `OpenBSD/usr.sbin/unbound` and `OpenBSD/sbin/unwind/libunbound`. There was considerable noise due to the cryptography and parsing-related functions included in the original `Unbound` codebase. The clusters identified by TIVER for the `Unbound` group were shown in Listing 2.

Listing 2: Directory structure of `OpenBSD` where reused code from `Unbound` exists.

```
1 OpenBSD/ // Clusters before removing noise
2 ⊢ usr.sbin/unbound/          [C1] (proportion: 0.582)
3 |  ⊢ rbtree.c (file) ...
4 ⊢ usr.sbin/nsd/              [C2] (proportion: 0.010)
5 |  ⊢ rbtree.c (file) ...
6 ⊢ sbin/unwind/libunbound/    [C3] (proportion: 0.398)
7 |  ⊢ rbtree.c (file) ...
8 ⊢ gnu/ ...                   [C4] (proportion: 0.008)
9 ⊢ lib/libc/crypt/ ...        [C5] (proportion: 0.001)
```

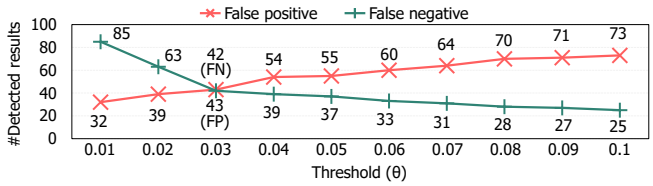


Fig. 3: Experimental results for measuring sensitivity of θ.

TIVER effectively identifies duplicate components and removes noise using code clustering. For instance, C1 and C2, which share the Level-1 node (`usr.sbin`), were separated owing to the unknown duplicate (`rbtree.c`). C1, with a high proportion of functions (0.582), was retained, while C2, with a minimal proportion (0.01), was eliminated as noise. C3 was kept for its significant reuse (0.398), while C4 and C5, containing mostly algorithmic code, were eliminated. Consequently, only clusters C1 and C3 remained, allowing TIVER to confirm the redundant reuse of `Unbound`. The adaptive version is identified for each cluster after noise pruning.

Threshold sensitivity. We used a θ value of 0.03 in our experiments. To measure threshold sensitivity, we investigated each noise elimination result of TIVER while increasing θ by 0.01 from 0.01 to 0.1.

Figure 3 presents the measurement results. We confirmed that as θ decreases, the possibility of eliminating noise decreases (*i.e.*, more FNs), and *vice versa*. Although the difference was not considerable, we selected the point where the balance between FN and FP was the most optimal. Specifically, when θ was set to 0.03, the balance between FN and FP was optimal, and notably, the total number of false results (*i.e.*, 85) was the lowest among all threshold values (*i.e.*, the F1-score was highest when θ was 0.03).

The fact that the accuracy of noise elimination does not change exponentially with different θ values demonstrates the overall high efficiency of our algorithm. Furthermore, we set θ to 0.03 in the experiment to achieve balanced results.

Answer to RQ2. TIVER demonstrates its effectiveness by (1) covering various versions that coexist in reused code regions, (2) identifying duplicate components, and (3) improving version identification accuracy by removing noise. The case study demonstrates TIVER's capability to identify adaptive versions even in complex reuse scenarios.

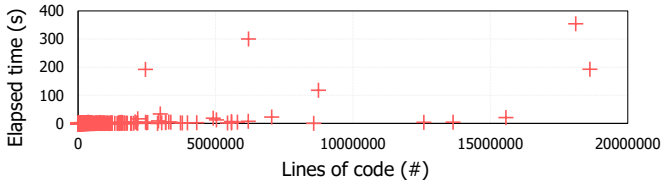


Fig. 4: Elapsed time for identifying adaptive versions in 869 popular C/C++ software programs with various code sizes.

C. Performance of TIVER

We assessed the performance and scalability of TIVER by measuring the time required to identify the adaptive version across target software sizes ranging from 1K to 20M lines of code (excluding the CENTRIS execution time). Figure 4 shows the results of the elapsed time measurements. We confirmed that TIVER can quickly identify the adaptive version of OSS components, even as the target software code size increases. Specifically, TIVER identified the adaptive version within an average of 1.67 s per target software (median of 0.03 s). While TIVER took slightly longer for some OSS with a very large number of components or extensive code size, it successfully identified the adaptive version within 30 s for all but five target software programs. The rapid performance of TIVER suggests that it can be effectively used for OSS management in practice.

Answer to RQ3. TIVER identified the adaptive version of the OSS components within an average of 1.67 s (*i.e.*, fast enough), without being significantly affected by the code size of the target software (*i.e.*, scalable).

D. Practicality of TIVER

TIVER can be effectively utilized in vulnerability management, which is particularly critical in supply chain security.

Methodology. For the 1,918 OSS components identified in Section IV-A, we initially discovered vulnerabilities using CENTRIS (*i.e.*, a single version-based approach), referencing the CPE of the NVD to extract vulnerabilities associated with the identified OSS and version [12]. We then used TIVER to examine the FPs and FNs of this approach, focusing on the vulnerabilities associated with the reused OSS and version.

1. **Extract vulnerable functions.** We extract functions associated with CVEs (*i.e.*, *vulnerable functions*) identified through the single version-based approach [8], [24], [29]
2. **Check function reuse.** TIVER then determines whether the vulnerable function has been reused in the target program. First, TIVER checks whether a function with the same name as the vulnerable function exists in the target software. If none is found, it searches for similar code using TLSH hashes stored by CENTRIS, which enable the detection of similar hashes [4]. If the vulnerable function has not been reused, TIVER defines this as an FP.
3. **Verify function version.** For components with multiple versions, TIVER examines each reused function's version. If the vulnerable function is reused but its version is not affected by the CVE, TIVER marks it as an FP.

TABLE VI: The practicality of TIVER in vulnerability verification and management (for 1,918 OSS components).

Category	Count
• #Discovered vulnerable functions (single version-based)	2,267
- #FPs identified by TIVER	
- #FPs caused by unused code	1,359
- #FPs caused by function updates	488
+ #FNs that can be overcome by TIVER	507
• Total vulnerable functions verified by TIVER	927

4. **Identify missed vulnerabilities.** Finally, TIVER iterates through all CVEs for the OSS, extracting vulnerable functions. If a vulnerable function (1) is reused in the target program, (2) belongs to an affected version, and (3) was missed by the single version-based approach, TIVER identifies it as an FN.

Result. Table VI summarizes the practicality evaluation results. Among the 1,918 OSS components, the single version-based approach initially discovered 2,267 vulnerable functions (associated with 314 CVE IDs) across 160 components. This high number was due to testing on older versions for a fair comparison with CENTRIS.

Using TIVER, more advanced vulnerability management was achieved. TIVER identified:

- 1,359 FPs where vulnerable functions were not reused.
- 488 FPs where vulnerable functions were reused but updated to safe versions.

In total, TIVER eliminated 1,847 FPs (81.47%) from the 2,267 functions identified by the single version-based approach. Manual verification of approximately 10% of the FPs confirmed that these vulnerable functions were either not reused or had been patched (*i.e.*, correct FPs).

In addition, TIVER identified 507 previously missed vulnerable functions (*i.e.*, FNs) that were overlooked by the single version-based approach because the overall version was determined based on the version to which the majority of reused functions belong. These vulnerabilities were difficult to detect without TIVER's finer-grained function-level versioning.

In conclusion, TIVER demonstrated its practicality by leveraging fine-grained versioning and adaptive versions to significantly reduce both FPs and FNs, enhancing vulnerability verification and management.

Case study. `FreeBSD` (v12.2.0) reuses `OpenSSH`, primarily based on version V_7_9_P1, but incorporates over 10 different versions within the reused code. The SBOM for `FreeBSD` can appear as follows (using the CycloneDX [30] format).

First, TIVER can identify areas where OSS has been reused, thereby contributing to supply chain security. For example, before pruning (see Listing 4), TIVER detected reused `OpenSSH` functions spanning versions 2.5.2 to 8.0.1. While existing approaches [4], [12] considered all these versions, TIVER accurately pruned noisy versions, identifying the adaptive version as *7.8.1. By specifying the version of each reused function (*i.e.*, lines #7, #10, and #14 in Listing 4), TIVER aids in SBOM creation and improves vulnerability detection.

Listing 3: Part of `FreeBSD`'s CycloneDX SBOM using the adaptive version identified by TIVER.

```
<?xml version="1.0" encoding="UTF-8"?>
<bom xmlns="http://cyclonedx.org/..." version="1">
 <metadata> ...
  <component type="operating-system">
   <name>FreeBSD</name>
   <version>12.2.0</version> ...
 <components>
  <component type="library">
  <name>OpenSSH</name>
  <version>⋆7.8.1</version> ...
```

Listing 4: Supplementary material for detailed OSS management includes the code areas where `OpenSSH` is reused in `FreeBSD` and the versions of each reused function (see Section III-E).

```
1 OpenSSH
2 (Before pruning) Identified adaptive version:  ⋆2.5.2
3 (After pruning) Cluster C1's adaptive version: ⋆7.8.1
4 ===
5 crypto/openssh [C1] // FreeBSD/crypto/openssh
6  ⊢ scp.c (file)
7  |  ⊢ sink (function): 8.0.1
8  ...
9  ⊢ gss-genr.c (file)
10 |  ⊢ ssh_gssapi_check_mechanism (function): 7.8.1
11 ...
12  ⊢ contrib (directory)
13 |  ⊢ gnome-ssh-askpass2.c (file)
14 |  |  ⊢ passphrase_dialog (function): 8.0.1
15 ...
```

Using TIVER, more advanced vulnerability management is possible. For example, searching for vulnerabilities in `OpenSSH` V_7_9_P1 in the CVE report[3] returned 13 results [4], [12], but not all are relevant to `FreeBSD`.

- **FP example.** CVE-2018-20685 (incorrect authorization in the `sink` function, affecting up to `OpenSSH` 7.9.1) was flagged by existing approaches as affecting `FreeBSD`. However, TIVER identified the `sink` function as originating from `OpenSSH` 8.0.1 (line #7 in Listing 4), confirming the issue was resolved in `FreeBSD`.
- **FN example.** CVE-2018-15919 (information disclosure in `ssh_gssapi_check_mechanism`, affecting up to `OpenSSH` 7.8.1) was missed by existing approaches focused on version 7.9.1. TIVER identified this function as originating from `OpenSSH` 7.8.1 (line #10 in Listing 4), showing that the vulnerability remained unpatched in `FreeBSD` 12.2.0 but was addressed in later versions.

By identifying the functions harboring each vulnerability and validating the versions of the reused functions (a process that can be easily automated), TIVER can detect only propagated vulnerabilities.

Answer to RQ4. TIVER can be practically utilized to enhance supply chain security, particularly by improving the efficiency of vulnerability verification and management (*e.g.*, eliminating FPs, which account for 81.47% of the detection results from the single version-based approach).

[3]https://www.cvedetails.com/vulnerability-list/vendor_id-97/product_id-585/version_id-1295245/Openbsd-Openssh-7.9.html

V. DISCUSSION

A. Considering the latest versions of reused functions

When TIVER assesses the adaptive version for each cluster, it considers the latest versions of the included functions (see Section III-E). From the perspective of OSS management, our choice to consider the latest version can be justified. In the latest version among the versions that a function belongs to, vulnerabilities present in previous versions are resolved. If versions other than the latest one are considered, it may lead to the false impression that resolved vulnerabilities still exist, resulting in FPs. Hence, TIVER identifies the adaptive version by considering the latest versions of the reused functions.

B. Using filenames as indicators for clustering

TIVER identifies duplicate components by examining reused file names. While we initially used function source code for finer granularity, this approach increased false alarms. For example, when different versions of OSS are redundantly reused, the same function may have different syntax. Addressing this requires similarity-based matching, but this frequently misidentifies functions with similar syntax as identical, significantly reducing clustering accuracy. We determined that this case occurs far more frequently and causes a greater loss in accuracy compared to changes in file names.

Specifically, when we investigated cases where file names had changed, in our setup, this accounted for 0.6% of the total (2,558 out of 433,485 reused files). In such cases, TIVER may produce incorrect results in clustering. However, this is a rare case, and thus we decided that utilizing file names has advantages compared to other levels of granularity. Our experiments demonstrated that the file name-based approach is sufficiently effective (see Section IV-A).

C. Applications of TIVER

Including the aspects mentioned throughout the paper, TIVER can be applied in various ways to enhance supply chain security. For example, by identifying reused code areas and determining the version of each function, TIVER can pinpoint the modified sections, which can contribute to more efficient SBOM generation. Furthermore, when multiple products in a product line reuse the same OSS, TIVER can reduce management complexity by tracking the version and code areas of the OSS used in specific product versions. This can help prevent conflicts between product variations.

D. Threats to validity

To demonstrate TIVER's generality, we conducted experiments using 10,417 OSS components and 2,025 popular software programs. While this dataset provides valuable insights, it may not fully represent the entire OSS ecosystem. Next, due to the lack of ground truth, two annotators manually analyzed TIVER's detection results. Most cases (95.6%) were clearly classified, with conflicts in only 84 out of 1,918 cases, resolved through discussion. However, differences in experience between the annotators introduced potential bias,

as most conflicts were resolved in favor of the more experienced analyst. Next, the concept of adaptive version was first introduced in TIVER, making direct comparisons with existing research infeasible. Comparisons with CNEPS and V1SCAN aimed to demonstrate TIVER's effectiveness, not critique prior methods. In addition, in the experiment measuring the practicality of TIVER, if the CPE provides incorrect vulnerable version information, our results may contain minor errors. Finally, while efforts were made to align collection dates of software and OSS datasets, slight discrepancies may have caused minor differences between identified and actual reused versions.

E. Limitations and future work

Despite the significant contributions of TIVER, several limitations should be acknowledged. First, TIVER can only identify adaptive versions when the source code of the target software is provided. Next, although TIVER demonstrates high accuracy, it struggles to correctly distinguish duplicate components where there are no overlapping files among each reused code region. Simply considering the finer granularity does not address this issue, because these are more frequently changed during the OSS reuse process than file names (see Section V-B). Therefore, we plan to investigate methods to utilize alternative granularities and methods to resolve this challenge. In addition, a reused function may not belong to any version of the OSS due to developers' custom-modifications. When identifying the version of a reused function, we utilized the CENTRIS dataset, which mapped functions that did not belong to any version of the OSS to the version with the most similar function. From a supply chain security perspective, custom-modified functions can also have considerable implications. Because this issue is more of a policy matter than a technical limitation and represents a tiny portion (less than 1% in our experiment), we did not consider it in the current study; if it becomes an issue, we plan to conduct research on this aspect as well. Finally, TIVER cannot be applied in cases of OSS reuse through package managers or the reuse of compiled libraries (*e.g.*, ".so" or ".dll"), as it does not handle these scenarios.

VI. RELATED WORK

Many studies have been proposed to identify reused third-party OSS components (*e.g.*, [2]–[4], [11], [16], [31]–[36]). For example, CENTRIS [4] utilizes code segmentation for identifying modified OSS components. LibD [32] employs a feature hashing technique to detect reused third-party libraries. CNEPS [16] attempted to examine dependencies among OSS components. OSSFP [20] attempted to detect third-party components by generating unique fingerprints of OSS. While OSSFP tried to remove noise, this was noise in component detection rather than in version identification, which could lead to necessary code for OSS version identification being mistakenly identified as noise and removed. Furthermore, they failed to consider duplicate components and version diversity.

Although these approaches can contribute to supply chain security by identifying third-party libraries, there has been no existing approach that effectively distinguishes duplicate components (see Section IV-A) and removes noise, which are prerequisites for identifying the adaptive version.

Several approaches have attempted to identify the versions of third-party libraries (*e.g.*, [2], [4], [12], [14], [35], [37]–[41]). CENTRIS [4] and V1SCAN [12] identify the predominant version of each component as the main version. BinaryAI [35] and LIBVDIFF [14] operate at the binary level to identify OSS components and map them to specific versions. VES [37] extracts a version in binary as a single string, utilizing data-flow analysis. OSSPOLICE [2] and LIBRARIAN [39] also assign a single version to identified OSS components. LIBDB [40] adopts a function call graph-based comparison to detect third-party libraries, FIRMSEC [41] determine the OSS version with the number of matched features.

These approaches can be utilized for supply chain security through vulnerable version identification. However, they fail to overcome the version diversity issue addressed in this paper. Moreover, they lack methods to effectively distinguish duplicate components and eliminate noise. Therefore, they cannot be applied to solve our target problem.

VII. CONCLUSION

Software reuse has led to the coexistence of multiple versions of OSS functions within target programs, making version identification and security management challenging. In this regard, we present TIVER, a novel approach for identifying adaptive versions of C/C++ OSS components, utilizing two key techniques: finer-grained versioning and OSS code clustering. Our experiments have demonstrated that TIVER effectively identifies adaptive versions while distinguishing duplicate components and eliminating noise. TIVER can be used to perform effective vulnerability management and enhance supply chain security.

DATA AVAILABILITY

The source code of TIVER and the package for replicating the experimental results are available in the public repository: https://github.com/Genius-Choi/TIVER-public.

ACKNOWLEDGMENT

We appreciate anonymous reviewers for their valuable comments. This work was supported by Institute of Information & Communications Technology Planning & Evaluation (IITP) grant funded by the Korea government(MSIT) (No.RS-2022-II220277, Development of SBOM Technologies for Securing Software Supply Chains, and No.RS-2024-00440780, Development of Automated SBOM and VEX Verification Technologies for Securing Software Supply Chains). In addition, this research was supported by Culture, Sports and Tourism R&D Program through the Korea Creative Content Agency grant funded by the Ministry of Culture, Sports and Tourism in 2024 (Project Name: International Collaborative Research and Global Talent Development for the Development of Copyright Management and Protection Technologies for Generative AI, Project Number: RS-2024-00345025).

REFERENCES

[1] C. V. Lopes, P. Maj, P. Martins, V. Saini, D. Yang, J. Zitny, H. Sajnani, and J. Vitek, "DéjàVu: a map of code duplicates on GitHub," in *Proceedings of the ACM on Programming Languages*, vol. 1, no. (OOPSLA). ACM, 2017, p. 84.

[2] R. Duan, A. Bijlani, M. Xu, T. Kim, and W. Lee, "Identifying Open-Source License Violation and 1-day Security Risk at Large Scale," in *Proceedings of the 24th ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2017, pp. 2169–2185.

[3] X. Zhan, T. Liu, L. Fan, L. Li, S. Chen, X. Luo, and Y. Liu, "Research on Third-Party Libraries in Android Apps: A Taxonomy and Systematic Literature Review," *IEEE Transactions on Software Engineering*, 2021.

[4] S. Woo, S. Park, S. Kim, H. Lee, and H. Oh, "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)*, 2021, pp. 860–872.

[5] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou, and W. Shi, "MVP: Detecting Vulnerabilities using Patch-Enhanced Vulnerability Signatures," in *Proceedings of the 29th USENIX Security Symposium (Security)*, 2020, pp. 1165–1182.

[6] S. Woo, D. Lee, S. Park, H. Lee, and S. Dietrich, "V0Finder: Discovering the Correct Origin of Publicly Reported Software Vulnerabilities," in *Proceedings of the 30th USENIX Security Symposium (Security)*, 2021, pp. 3041–3058.

[7] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, and X. Peng, "Demystifying the Vulnerability Propagation and Its Evolution via Dependency Trees in the NPM Ecosystem," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 672–684.

[8] S. Woo, H. Hong, E. Choi, and H. Lee, "MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components," in *Proceedings of the 31st USENIX Security Symposium (Security)*, 2022, pp. 3037–3053.

[9] Y. Wu, Z. Yu, M. Wen, Q. Li, D. Zou, and H. Jin, "Understanding the Threats of Upstream Vulnerabilities to Downstream Projects in the Maven Ecosystem," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1046–1058.

[10] Y. Gu, L. Ying, Y. Pu, X. Hu, H. Chai, R. Wang, X. Gao, and H. Duan, "Investigating package related security threats in software registries," in *2023 IEEE Symposium on Security and Privacy (SP)*, 2023.

[11] X. Zhan, L. Fan, S. Chen, F. Wu, T. Liu, X. Luo, and Y. Liu, "ATVHunter: Reliable Version Detection of Third-Party Libraries for Vulnerability Identification in Android Applications," in *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1695–1707.

[12] S. Woo, E. Choi, H. Lee, and H. Oh, "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)*, 2023, pp. 6541–6556.

[13] A. Dann, B. Hermann, and E. Bodden, "UPCY: Safely Updating Outdated Dependencies," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 233–244.

[14] C. Dong, S. Li, S. Yang, Y. Xiao, Y. Wang, H. Li, Z. Li, and L. Sun, "LibvDiff: Library Version Difference Guided OSS Version Identification in Binaries," in *Proceedings of the 46th International Conference on Software Engineering (ICSE)*, 2024, pp. 791–802.

[15] W. Tang, Z. Xu, C. Liu, J. Wu, S. Yang, Y. Li, P. Luo, and Y. Liu, "Towards Understanding Third-party Library Dependency in C/C++ Ecosystem," in *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, 2022, pp. 1–12.

[16] Y. Na, S. Woo, J. Lee, and H. Lee, "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)*, 2024, pp. 2918–2929.

[17] R. Croft, M. Babor, and H. Chen, "Noisy label learning for security defects," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2024, pp. 435–447.

[18] F. Wallner, B. K. Aichernig, and C. Burghard, "It's Not a Feature, It's a Bug: Fault-Tolerant Model Mining from Noisy Data," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2024, pp. 327–339.

[19] B. Liang, P. Bian, Y. Zhang, W. Shi, W. You, and Y. Cai, "AntMiner: mining more bugs by reducing noise interference," in *2024 IEEE/ACM 46th International Conference on Software Engineering (ICSE)*. IEEE/ACM, 2016, pp. 333–344.

[20] J. Wu, Z. Xu, W. Tang, L. Zhang, Y. Wu, C. Liu, K. Sun, L. Zhao, and Y. Liu, "OSSFP: Precise and Scalable C/C++ Third-Party Library Detection using Fingerprinting Functions," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 270–282.

[21] N. Telecommunications and I. Administration, "NTIA Software Component Transparency with SBOM (Software Bill of Materials)," 2024, https://www.ntia.doc.gov/SoftwareTransparency.

[22] C. W. Krueger, "Software reuse," in *ACM Computing Surveys (CSUR)*, vol. 24, no. 2. ACM, 1992, pp. 131–183.

[23] A. Decan and T. Mens, "What Do Package Dependencies Tell Us About Semantic Versioning?" *IEEE Transactions on Software Engineering (TSE)*, vol. 47, no. 6, pp. 1226–1240, 2019.

[24] S. Kim, S. Woo, H. Lee, and H. Oh, "VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery," in *Proceedings of the 38th IEEE Symposium on Security and Privacy (SP)*, 2017, pp. 595–614.

[25] C. team, "Dataset for CENTRIS," Jul. 2024.

[26] NPM, "Node Package Manager," 2024, https://www.npmjs.com/.

[27] Ctags, "Universal Ctags," 2024, https://github.com/universal-ctags/ctags.

[28] Anytree, "Simple, lightweight and extensible Tree data structure." 2024, https://github.com/c0fec0de/anytree.

[29] S. Woo, E. Choi, and H. Lee, "A large-scale analysis of the effectiveness of publicly reported security patches," *Computers & Security*, p. 104181, 2024.

[30] OWASP, "CycloneDX," https://cyclonedx.org/.

[31] M. Backes, S. Bugiel, and E. Derr, "Reliable Third-Party Library Detection in Android and its Security Applications," in *Proceedings of the 23rd ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2016, pp. 356–367.

[32] M. Li, W. Wang, P. Wang, S. Wang, D. Wu, J. Liu, R. Xue, and W. Huo, "LibD: Scalable and Precise Third-party Library Detection in Android Markets," in *Proceedings of the 39th International Conference on Software Engineering (ICSE)*. IEEE, 2017, pp. 335–346.

[33] Z. Ma, H. Wang, Y. Guo, and X. Chen, "LibRadar: Fast and Accurate Detection of Third-party Libraries in Android Apps," in *Proceedings of the 38th International Conference on Software Engineering: Companion (ICSE-Companion)*, 2016, pp. 653–656.

[34] W. Tang, D. Chen, and P. Luo, "BCFinder: A Lightweight and Platform-independent Tool to Find Third-party Components in Binaries," in *Proceedings of the 25th Asia-Pacific Software Engineering Conference (APSEC)*. IEEE, 2018, pp. 288–297.

[35] L. Jiang, J. An, H. Huang, Q. Tang, S. Nie, S. WU, and Y. Zhang, "BinaryAI: Binary Software Composition Analysis via Intelligent Binary Source Code Matching," in *Proceedings of the 46th International Conference on Software Engineering (ICSE)*, 2024, pp. 2771–2783.

[36] X. Xu, Q. Zheng, Z. Yan, M. Fan, A. Jia, and T. Liu, "Interpretation-enabled software reuse detection based on a multi-level birthmark model," in *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, 2021.

[37] X. Hu, W. Zhang, H. Li, Y. Hu, Z. Yan, X. Wang, and L. Sun, "VES: A Component Version Extracting System for Large-Scale IoT Firmwares," in *Wireless Algorithms, Systems, and Applications: 15th International Conference, WASA 2020*. ACM, 2020, pp. 39–48.

[38] W. Zhang, Y. Chen, H. Li, Z. Li, and L. Sun, "PANDORA: A Scalable and Efficient Scheme to Extract Version of Binaries in IoT Firmwares," in *IEEE International Conference on Communications (ICC)*. IEEE, 2018, pp. 1–6.

[39] S. Almanee, A. Ünal, M. Payer, and J. Garcia, "Too Quiet in the Library: An Empirical Study of Security Updates in Android Apps' Native Code," in *Proceedings of the 43rd International Conference on Software Engineering (ICSE)*, 2021, pp. 1347–1359.

[40] W. Tang, Y. Wang, H. Zhang, S. Han, P. Luo, and D. Zhang, "LibDB: an effective and efficient framework for detecting third-party libraries in binaries," in *Proceedings of the 19th International Conference on Mining Software Repositories (MSE)*, 2022, pp. 423–434.

[41] B. Zhao, S. Ji, J. Xu, Y. Tian, Q. Wei, Q. Wang, C. Lyu, X. Zhang, C. Lin, J. Wu, and R. Beyah, "A large-scale empirical analysis of the vulnerabilities introduced by third-party components in IoT firmware," in *ISSTA 2022: Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2022, pp. 442–454.