VulSCA: A Community-Level SCA Approach for Accurate C/C++ Supply Chain Vulnerability Analysis

Yutao Hu[*,†], Chaofan Li[*], Yueming Wu[*,†,§], Yifeng Cai[‡], Deqing Zou[*,†]
[*]National Engineering Research Center for Big Data Technology and System
Services Computing Technology and System Lab, Hubei Engineering Research Center on Big Data Security
School of Cyber Science and Engineering, Huazhong University of Science and Technology, China
[†]Jinyinhu Laboratory, China
[‡]MOE Key Lab of HCST (PKU), School of Computer Science, Peking University, China
[§]Corresponding author

***Abstract*—With the widespread adoption of *third-party libraries* (TPLs) in C/C++ development, software supply chain security has become critical. Existing C/C++ supply chain vulnerability analysis approaches have notable limitations. Some focus exclusively on dependency identification, leading to *false positives* (FPs), while others emphasize vulnerability detection but ignore dependencies, requiring costly full-repository scans that hinder rapid response to supply chain vulnerabilities. To address this, we explore an appropriate granularity for accurate dependency construction and vulnerability detection. We propose a community-level *software composition analysis* (SCA) approach that models the project's call graph as a social network and applies community detection. Dependencies between projects and TPLs are then established through community similarity. For vulnerability detection, we perform clone-based detection within dependent communities to verify the existence of vulnerabilities, and introduce a two-stage reachability analysis to determine whether they can propagate to the target project. We implement *VulSCA*, the first C/C++ SCA framework that integrates both vulnerability detection and reachability analysis. Experimental results show that *VulSCA* outperforms *CENTRIS* and *OSSFP* in SCA with a 4–12% improvement in F1-score. In supply chain vulnerability detection, it achieves 44–48% higher F1-scores than version-based methods and 17–23% higher than code-based methods. In terms of efficiency, *VulSCA* incurs lower overall overhead than all code-based approaches. Furthermore, *VulSCA* identifies 32 previously unpatched supply chain vulnerabilities in widely used open-source projects, which have already been reported to the respective vendors.**

I. Introduction

Ensuring the security of the software supply chain has become increasingly critical, particularly in C/C++ development where *third-party libraries* (TPLs) are extensively adopted. While *software composition analysis* (SCA) is essential for component identification and dependency management, constructing accurate supply chains in C/C++ remains difficult due to the diversity of TPL integration methods and the lack of a standardized dependency management system. Unlike conventional vulnerabilities, supply chain vulnerabilities can propagate across projects and ecosystems, enabling attackers to steal sensitive data or disrupt downstream operations. Thus, achieving accurate and efficient detection of supply chain vulnerabilities remains an urgent and unresolved challenge.

Existing approaches. Methods for detecting supply chain vulnerabilities in C/C++ software can be broadly classified into two categories: version-based and code-based [1]. ***(1)*** *Version-based methods* (*i.e.,* traditional SCA techniques) [2], [3], [4], [5] identify dependencies by analyzing project-level features. Once a dependency is detected, any known vulnerabilities associated with the corresponding TPL version are directly flagged as supply chain vulnerabilities. ***(2)*** *Code-based methods* (*i.e.,* clone-based vulnerability detection techniques) [3], [6] search the target codebase against a database of vulnerable code snippets. If a segment closely matches a known vulnerable function, it is reported as a potential supply chain risk. In practice, version-based methods focus solely on dependency identification and overlook whether the vulnerable code is truly present, whereas code-based methods emphasize vulnerability detection but ignore dependency context.

Limitations of existing approaches. Most version-based methods seek to improve SCA accuracy and efficiency by designing project-level features and inferring dependencies through feature similarity. However, these features are generally unrelated to vulnerability semantics. For example, *OSSPolice* [2] relies on directory structures, while *CENTRIS* [7] leverages shared original functions across projects. As a result, these methods often incur numerous false negatives (FNs) and FPs in supply chain vulnerability detection.

Conversely, code-based approaches focus on improving clone detection precision but lack pre-built dependency relationships, which forces exhaustive repository scans. This limitation reduces their responsiveness to newly disclosed threats. Moreover, these methods merely confirm the presence

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.230613
www.ndss-symposium.org

of vulnerable code without verifying its execution reachability, leading to FPs.

Our Approach. To overcome the limitations of existing methods, we propose a new analysis granularity that not only accurately captures project–TPL dependencies but also effectively supports supply chain vulnerability detection and reachability analysis. Specifically, we model the project's call graph as a social network and introduce a ***semantic community***-level SCA approach tailored for C/C++ software.

We begin by extracting the project's call graph and applying community detection to cluster semantically related functions into semantic communities. Compared with traditional function- or file-level granularity, semantic communities better reflect the modularity and logical organization of the code, offering a more coherent unit for dependency and vulnerability analysis. For each community, we extract two sets of features: structure features that represent inter-function call relationships, and code features that capture the textual and semantic characteristics. We then assess project–TPL dependencies by measuring the similarity between their semantic communities.

To detect reachable supply chain vulnerabilities, we first apply clone vulnerability detection within the identified dependent communities to determine whether known vulnerable code is actually reused in the project. However, even when the vulnerable code is confirmed to exist, it remains uncertain whether it is reachable, meaning whether the vulnerable code can actually be executed during the project's operation. Inspired by prior work [8], we define a vulnerability as reachable if it can be triggered from the project's entry point. Specifically, we construct a community-level call graph and perform a two-stage reachability analysis: inter-community analysis identifies vulnerable communities that are potentially reachable, followed by intra-community analysis to determine whether specific vulnerable functions are truly executable. This layered strategy balances precision and scalability, mitigating the path explosion problem that typically occurs in whole-project analysis.

Evaluation. We implement *VulSCA*, the first C/C++ SCA method that integrates both supply chain vulnerability detection and reachability analysis. For evaluation, we construct a ground truth dataset from real-world software dependencies and reachable supply chain vulnerabilities. In the SCA task, *VulSCA* achieves a 4–12% improvement in F1-score compared with state-of-the-art C/C++ SCA tools, namely *CENTRIS* [7] and *OSSFP* [3]. In the supply chain vulnerability detection task, *VulSCA* incurs lower overhead than all code-based methods. It also surpasses version-based approaches (*CENTRIS* and *OSSFP*) with a 44–48% gain in F1-score, and code-based approaches (*Vuddy* [4], *Fire* [5], and *AntMan* [9]) with a 17–23% gain. Finally, by analyzing widely used open-source projects, *VulSCA* uncovered 32 previously unpatched supply chain vulnerabilities, all of which have been responsibly reported to vendors.

Contributions. This work makes the following contributions:

- We introduce a novel granularity, the semantic community, for effective C/C++ SCA. By extracting call graphs, performing community detection, and combining structural and code-level features, we identify dependencies between projects and TPLs through community similarity.
- We propose a community-based supply chain vulnerability detection approach that first conducts vulnerability existence analysis to locate potentially vulnerable communities, and then performs inter- and intra-community reachability analysis to determine vulnerabilities that may pose real threats to the target project.
- We implement *VulSCA*[1], the first C/C++ SCA method that integrates both supply chain vulnerability detection and reachability analysis.
- We conduct extensive experiments showing that *VulSCA* consistently outperforms state-of-the-art tools. In the SCA task, it surpasses *OSSFP* and *CENTRIS* with a 4–12% improvement in F1-score. In vulnerability detection, it achieves a 17–48% F1-score improvement over traditional methods.

II. Background

A. Terminology Definition

This section defines the terminology used in this paper, clarifying their roles and relationships across the technical workflow: from foundational components (*i.e.,* TPL) to identification (*i.e.,* TPL detection), systematic analysis (*i.e.,* SCA), and security assessment (*i.e.,* supply chain vulnerability detection).

• **Third-Party Library (TPL).** A TPL is a reusable software component developed externally, typically open-source and hosted on platforms such as GitHub. TPLs significantly improve development efficiency but may introduce security risks through vulnerability propagation.

• **TPL Detection.** TPL detection identifies the TPLs integrated into a project, including their name, version, and location. Detection methods vary depending on the reuse approach: 1) Standardized reuse via package managers enables identification by parsing dependency files such as `pom.xml` or `requirements.txt`; 2) Non-standard reuse via source code copying requires detection techniques such as code comparison, feature extraction, or feature similarity analysis.

• **Software Composition Analysis (SCA).** SCA provides a comprehensive inventory and evaluation of TPLs in a project to assess security, license compliance, and quality. TPL detection forms the foundation of SCA, directly influencing downstream tasks such as license auditing and vulnerability tracking.

• **Supply Chain Vulnerability Detection.** This refers to identifying N-day vulnerabilities introduced through TPLs due to defects in TPL code. A project is considered vulnerable if it invokes a vulnerable function within a TPL. This detection relies heavily on the dependency graph constructed by SCA, which narrows the analysis scope and accelerates security response, as the graph can be pre-built and reused. When a new TPL vulnerability is disclosed, affected projects can be rapidly identified without scanning the entire codebase. For

[1] https://github.com/VulSCA/VulSCA

example, using a pre-built dependency graph, the number of functions to analyze in the *NGINX* [10] project decreases from 130,000 to about 2,000, yielding nearly a 60-fold speed up.

B. Semantic Community Definition

To improve the efficiency and accuracy of SCA and vulnerability reachability analysis, we introduce *semantic communities* as the basic analysis unit.

• **Community Detection.** A core method in complex network research used to discover densely linked node clusters [11]. Applied to function call graphs, it partitions functions into groups with strong internal coupling and frequent interactions, revealing modules.

• **Semantic Community.** A subgraph of functions that collaboratively implement specific functionality. Identified based on call frequency and coupling strength, semantic communities better reflect the code's logical organization than traditional function- or file-level granularity. While structurally similar to subgraphs, semantic communities differ from subgraph isomorphism or clone detection in several ways:

○ Composition: Formed by structurally coupled and semantically aggregated functions.

○ Characteristics: Strong internal cohesion, clear boundaries between communities, and high semantic consistency.

○ Practical alignment: Their modular structure aligns naturally with common software development practices, making them an ideal unit for SCA.

In our work, semantic communities enhance dependency detection accuracy and reduce call graph complexity, providing a scalable foundation for vulnerability reachability analysis.

C. Motivation

Unlike languages that rely on package managers, C/C++ projects often integrate TPLs by directly copying source code. This code-level reuse introduces unique challenges for SCA, especially in supply chain vulnerability detection.

To illustrate the limitations of existing SCA techniques in this context, we conduct a case study on *SPlayer* [12], an open-source audio player containing approximately 7.49 million lines of code. We analyze *SPlayer* using our approach, *VulSCA*, and compare its accuracy against representative version-based and code-based methods.

As shown in Fig. 1 (top), *SPlayer* incorporates selected functions from *OpenSSL* [13]. However, only a small subset of these functions is actually invoked by the core logic, as revealed by the function call graph. We use three representative vulnerabilities (CVE_1, CVE_2, and CVE_3) to highlight detection gaps. While CVE_2 affects a function that is executed by *SPlayer*, CVE_3 resides in a cloned but unreachable function, and CVE_1 is located in a function that is not reused at all.

Code-based Approaches. These methods detect code clones and report vulnerabilities in reused code as supply chain risks. However, without reachability analysis, they often yield FPs. For example, CVE_3 is flagged even though the function is unreachable. Through manual inspection, we identified 77 *OpenSSL*-related vulnerabilities in *SPlayer*, but only 25 are actually executed. As shown in Table I, tools such as *Vuddy* [4], *Fire* [5], and *AntMan* [9] achieve low precision (40%, 31%, and 48%, respectively), underscoring the need for ***vulnerability reachability analysis***. While integrating function-level reachability analysis could mitigate this issue, it is computationally expensive: the large number of functions and the complex call relationships in real-world projects lead to path explosion. Moreover, code-based approaches require clone detection across all functions of *SPlayer*, further reducing the timeliness of security response.

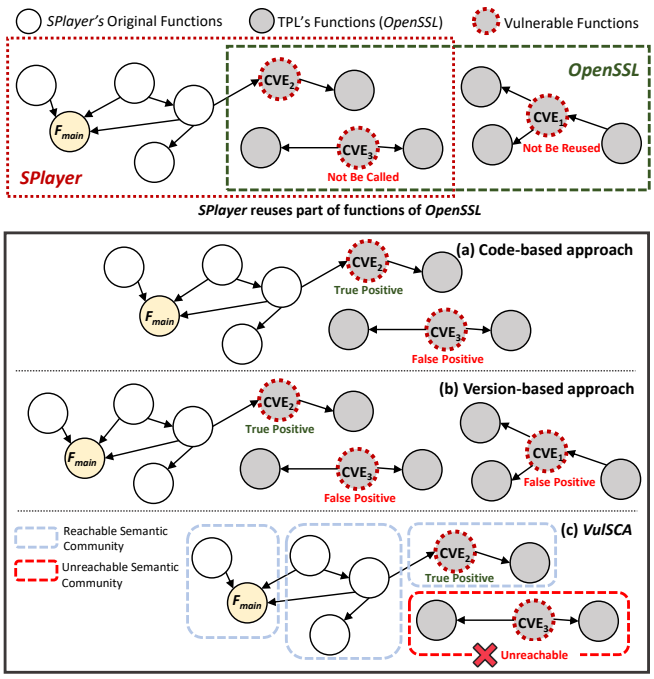

Fig. 1. An Example of *SPlayer* Reusing *OpenSSL*

TABLE I
VULNERABILITY DETECTION RESULTS FOR *SPlayer* USING *VulSCA*, VERSION-BASED, AND CODE-BASED APPROACHES

Group	Method	TP	FP	Precise	Recall	F1-score
Version-based	*CENTRIS*	14	245	0.05	0.56	0.10
	OSSFP	13	38	0.25	0.52	0.34
Code-based	*Fire*	22	50	0.31	0.88	0.45
	Vuddy	21	31	0.40	0.84	0.55
	Antman	20	22	0.48	0.80	0.69
VulSCA		19	3	0.86	0.76	0.81

Version-based Approaches. These tools analyze declared dependencies without examining the actual code reused. As a result, any known vulnerability in a dependent TPL is conservatively reported, regardless of whether the vulnerable function is present or invoked. In Fig.1 (b), both CVE_1 and CVE_3 are falsely reported. As shown in Table I, state-of-the-art tools like *CENTRIS* [7] and *OSSFP* [3] yield low precision (5% and 25%, respectively), due to their lack of ***vulnerability existence*** and ***reachability analysis***.

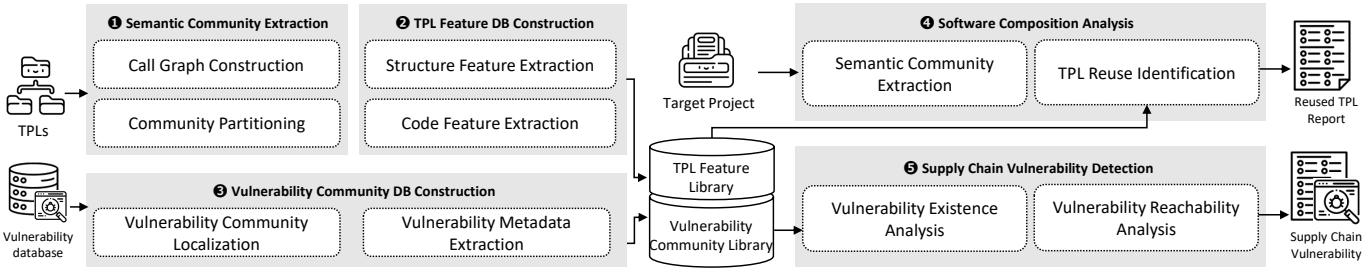

Fig. 2. High-level Workflow of *VulSCA*

Our Approach: *VulSCA*. Existing SCA methods fail to jointly perform vulnerability existence and reachability analysis, leading to high false-positive rates. To bridge this gap, we propose *VulSCA*, which introduces a novel granularity for SCA, *semantic communities*.

Each semantic community consists of a set of tightly coupled functions that collaborate to implement a specific functionality. For example, a "file-write" community might include `open_file()`, `write_file()`, and `close_file()`, all working together to perform file-writing. This granularity is motivated by our observation that developers typically reuse entire functional modules of a TPL rather than individual functions [14]. *VulSCA* therefore defines such modules as semantic communities. Accordingly, community-level dependencies more directly reflect the reuse patterns of C/C++ TPLs and provide effective support for SCA.

In addition, community-level analysis inherently offers advantages in supply-chain vulnerability detection. By analyzing dependencies at the community-level, *VulSCA* performs: ***1) Vulnerability Existence Analysis***: *VulSCA* determines whether a TPL vulnerability actually resides within a reused community. ***2) Vulnerability Reachability Analysis***: *VulSCA* conducts both inter- and intra-community reachability analysis to avoid the path-explosion problem inherent in function-level analysis. As shown in Table I, *VulSCA* achieves 86% precision, significantly reducing FPs and effectively bridging the gap between C/C++ SCA and supply-chain vulnerability detection.

III. Approach

To bridge C/C++ SCA with supply chain vulnerability detection, *VulSCA* defines a novel code analysis granularity, referred to as *semantic communities*.

A. System Overview

Fig. 2 illustrates the overall workflow of *VulSCA*, which consists of five stages. In analyzing TPLs, ***Stage 1*** partitions each TPL into semantic communities, and ***Stage 2*** extracts structure and code features from these communities to construct the TPL feature library. For known vulnerabilities, ***Stage 3*** associates each vulnerability with its corresponding semantic community, forming the vulnerability community library. In ***Stage 4***, *VulSCA* analyzes the target project by extracting its semantic community features and matching them against the TPL feature library to identify reused TPLs and corresponding communities. Finally, in ***Stage 5***, *VulSCA* checks whether any known vulnerable function exists in the project. It then performs both inter- and intra-community reachability analysis to evaluate whether these vulnerabilities can propagate into the project's core logic, thus reducing false positives.

B. Semantic Community Extraction

At this stage, *VulSCA* extracts the function call graph from each TPL and groups functions into communities based on call frequency and coupling strength. These semantic communities consist of tightly related functions and typically correspond to distinct functional modules.

1) Call Graph Construction

A semantic community is essentially a collection of functions with strong semantic coupling and frequent interactions. Its partitioning must rely on actual call relationships to ensure functional cohesion. Therefore, we first construct a function call graph for each project.

Call Graph Extraction. To make *VulSCA* resilient to minor code changes and to suppress noise from irrelevant code, we begin with source code normalization. After normalization, we extract the call graph from the updated code to achieve stable analysis. We use the static analysis tool *Doxygen* [15] to parse and extract call graphs of TPLs. Since our analysis must scale to large TPL databases and cannot afford compilation overhead, *Doxygen*'s static parsing capability, which operates without compilation, makes it an ideal choice.

In the resulting call graph, nodes represent functions and edges denote call relationships. We also compute a textual hash for each function to serve as its unique identifier.

Nested Dependency Elimination. C/C++ SCA is often complicated by nested dependencies among TPLs [7], where two distinct TPLs contain identical functions, causing the target project to depend on both and introducing false positives.

To resolve this issue, we adopt a two-step strategy based on function-creation timestamps and virtual node replacement. First, following *CENTRIS*, we use function creation timestamps to identify the canonical implementation: when a function appears in multiple TPLs, the instance with the earliest timestamp is retained as the original. Second, all non-original duplicates are replaced with *virtual nodes* in the call graph, eliminating redundancy while preserving the structural integrity of call relationships. Although virtual nodes do not

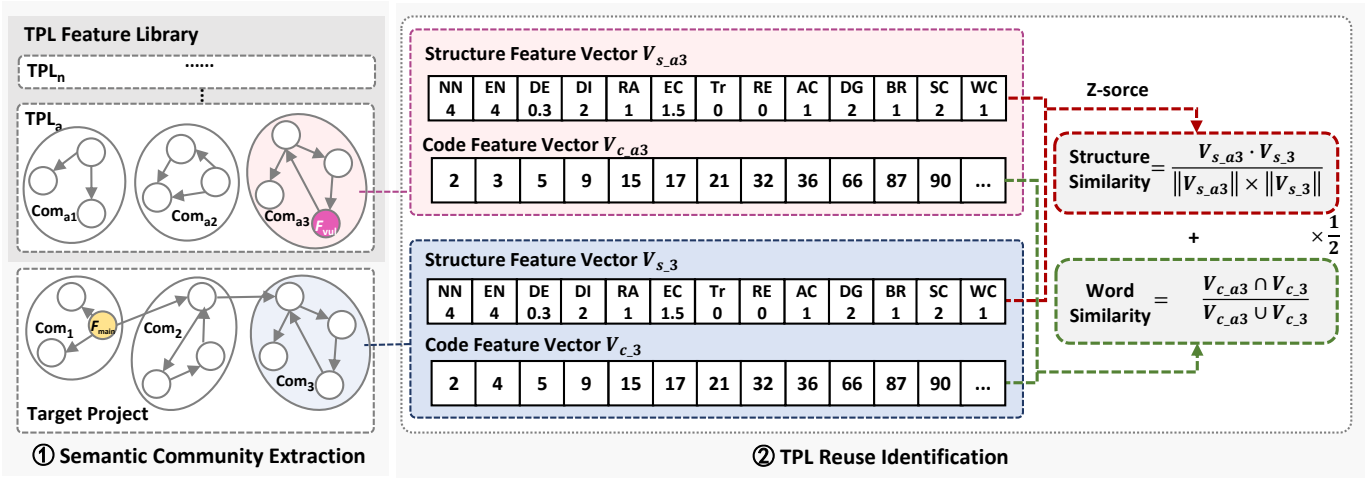

Fig. 3. An Example of Semantic Community Extraction and Software Composition Analysis

represent the TPL's core semantics, they still capture essential syntactic structure for *VulSCA*'s analysis.

2) Community Partitioning

TPLs typically provide extensive functionality, but real-world projects usually reuse only selected modules [14]. In C/C++ development, developers often clone entire feature units from TPLs, consisting of tightly coupled functions with frequent inter-calls. To capture this, *VulSCA* partitions the call graph into semantic communities, which are subgraphs of related functions implementing distinct features, as shown in Fig. 3. Compared with traditional function- or file-level partitioning, this approach better reflects functional boundaries and the modular structure of software.

For community partitioning, *VulSCA* models the call graph as a social network, where functions are nodes and calls are edges. By applying community detection algorithms, *VulSCA* groups frequently interacting and tightly coupled functions into cohesive modules (*i.e.,* semantic communities). We consider four widely used algorithms as candidates:

• *Louvain Algorithm* [16]: a greedy modularity optimization method that iteratively relocates nodes to maximize modularity, enabling fast detection in large networks.

• *Infomap Algorithm* [17]: formulates community detection as optimal compression of random-walk trajectories, enabling discovery of overlapping and hierarchical communities in complex networks.

• *Leiden Algorithm* [18]: an improved Louvain method that corrects community fragmentation, enhancing detection accuracy while retaining high efficiency, suitable for large-scale networks.

• *Greedy Modularity Algorithm* [19]: iteratively merges community pairs with the greatest modularity gain; simple in design but prone to local optima and less efficient on large networks, often used as a baseline.

After partitioning the call graph into communities, *VulSCA* filters out those unrelated to the project's core functionality, allowing SCA to focus on meaningful dependencies. As discussed in Section III-B1, some communities include *virtual nodes*, which usually originate from TPLs and do not represent the project's original semantics. Our empirical analysis shows that functions connected to these *virtual nodes* often invoke or wrap third-party functionality rather than core project logic. To reduce their impact, *VulSCA* discards communities in which *virtual nodes* constitute more than 50% of the nodes. For instance, a community containing only a *virtual node* and its caller is excluded, since the caller merely integrates a TPL feature instead of implementing original behavior. This filtering step ensures that only semantically relevant communities are preserved, thereby improving SCA accuracy.

C. TPL Feature DB Construction

At this stage, *VulSCA* extracts both structural and code features from each semantic community to build a lightweight TPL feature database. Structural features are derived from social network properties, capturing function-call relationships within the community and reflecting the organizational structure and logical flow of project features. Code features focus on the textual representation of functions, providing detailed insight into their concrete implementations.

1) Structure Feature Extraction

VulSCA extracts structural features by analyzing the function call graph of each semantic community. To capture call logic efficiently, we adopt social network properties. Based on the performance evaluation of *SNADroid* [20], we select the 11 fastest properties (No. 1–11 in Table II), each requiring under 0.001s. We further include the number of *weakly connected components* (WC), highlighted as important in *SNADroid*, along with its counterpart, the number of *strongly connected components* (SC), yielding 13 properties in total. These cover node-level, global, and connectivity-related aspects. For undirected properties, the community graph is treated as undirected, and for node-level properties, we use the average value across all nodes.

TABLE II
DETAILS OF COMMUNITY STRUCTURE FEATURES

No.	Social Network Property	Abbr.	Description	Feature Processing Time (s)
1	Number of Nodes	NN	The number of nodes in the network	<0.001
2	Number of Edges	NE	The number of edges in the network	<0.001
3	Density	DE	The density of the network	<0.001
4	Diameter	DI	The maximum eccentricity of the network	<0.001
5	Radius	RA	The minimum eccentricity of the network	<0.001
6	Eccentricity	EC	The maximum distance from a node to other nodes	<0.001
7	Reciprocity	RE	The ratio of bidirectional edges to all edges	<0.001
8	Algebraic Connectivity	AC	The second smallest eigenvalue of the *Laplacian* matrix	<0.001
9	Degree	DG	The number of edges connected to a node	<0.001
10	Bridges	BR	The number of bridges in the network	<0.001
11	Transitivity	TR	The fraction of closed triangles in the graph	<0.001
12	Strongly Connected Components	SC	The number of strongly connected components in the network	0.009
13	Weakly Connected Components	WC	The number of weakly connected components in the network	0.007

To account for scale differences among properties, we apply *Z-score normalization* [21], which standardizes each property relative to the mean and standard deviation. This produces a 1×13 structural feature vector, denoted V_s (see "Structure Feature Vector" in Fig. 3).

2) Code Feature Extraction

To complement structural metrics, *VulSCA* extracts text-based features to assess code similarity. We tokenize each function, aggregate tokens into a community-level vocabulary, and apply one-hot encoding to generate the "Code Feature Vector" V_c (see Fig. 3), enabling efficient similarity comparisons. Finally, *VulSCA* constructs the TPL feature library. Each semantic community is assigned a unique ID. The structural and code feature vectors, V_s and V_c, are linked to this ID to support community similarity matching during SCA. For vulnerability analysis, we also record the hash values of all functions in each community. Thus, each entry in the library contains information for one TPL, including its community IDs and, for each ID, the associated feature vectors and function hashes.

D. Vulnerability Community DB Construction

At this stage, *VulSCA* builds a community-level TPL vulnerability database to enable rapid identification of vulnerable dependencies in SCA results. The construction process consists of two steps: vulnerability community localization and vulnerability metadata extraction.

1) Vulnerability Community Localization

Known vulnerabilities, such as those reported in the NVD [22], are typically linked to commits that modify affected functions, which can thus be represented as vulnerable functions. *VulSCA* elevates the analysis granularity by mapping each vulnerable function to its enclosing semantic community. To achieve this, *VulSCA* computes a textual hash of each vulnerable function's source code as its unique identifier and searches the function-hash lists of all semantic communities for a match. The community containing the matching function is then identified as the *vulnerability community*.

2) Vulnerability Metadata Extraction

After localization, *VulSCA* constructs a community-level vulnerability database. Each identified vulnerability community is assigned a unique ID, which is then mapped to a standardized metadata unit encompassing all vulnerable functions within that community. Specifically, each vulnerability community ID corresponds to a hash set that stores the hash values of all known vulnerable functions in the community. This results in a structured entity where the community ID serves as the key and the associated function-hash list serves as the value. Such a standardized design provides a unified and efficient foundation for subsequent vulnerability detection.

E. Software Composition Analysis

As shown in Fig. 3, the core principle of *VulSCA*'s SCA is to infer dependencies by measuring the similarity between semantic communities of the target project and those of TPLs.

1) Semantic Community Extraction

When a project community closely matches a TPL community, *VulSCA* links only these two communities rather than associating the entire project with the TPL. In other words, *VulSCA* performs community-level instead of project-level SCA, thereby enabling more accurate supply-chain vulnerability detection.

For the target project under analysis, *VulSCA* applies the same procedure as for TPLs: it partitions the project's call graph into semantic communities and extracts their structural and code features. The detailed steps are consistent with those in Sections III-B2, III-C1, and III-C2.

2) TPL Reuse Identification

To identify dependencies between a project community and a TPL community, *VulSCA* leverages both the *Structure Feature Vector* (V_s) and the *Code Feature Vector* (V_c). *Cosine* similarity is used to measure structural similarity of V_s, while *Jaccard* similarity evaluates token-level similarity of V_c, as shown in Fig. 3. The combination of these metrics determines the overall semantic similarity between communities.

Because V_c is typically high-dimensional due to the large number of tokens in functions, evaluating it incurs higher computational cost. To improve efficiency, *VulSCA* first computes V_s similarity and applies an early filtering strategy: if V_s similarity falls below a threshold (0.5), the V_c comparison is skipped. Finally, *VulSCA* averages the Cosine and Jaccard scores to assess overall similarity. If the combined score

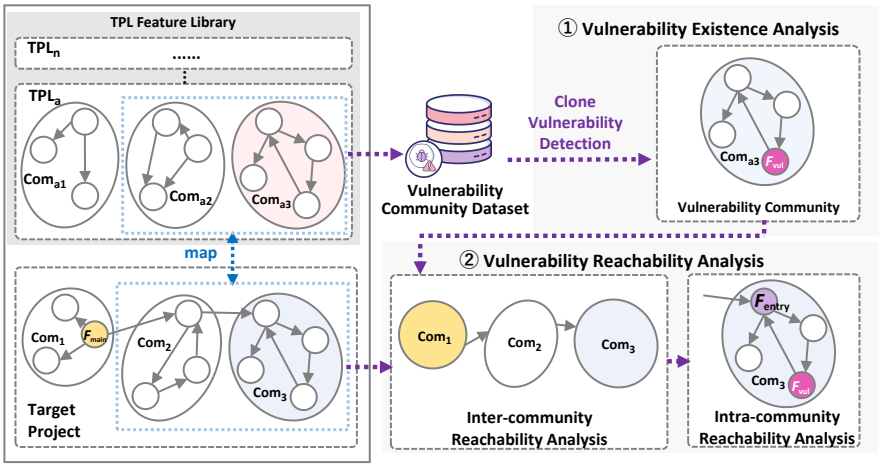

Fig. 4. An Example of Supply Chain Vulnerability Detection

exceeds a threshold θ, a reuse relationship is established, indicating that the target community depends on the TPL community. This community-level dependency further implies a version-level dependency between the corresponding projects.

F. Supply Chain Vulnerability Detection

At this stage, *VulSCA* first performs vulnerability existence analysis to determine whether known vulnerabilities are present in the reused TPL communities. It then conducts reachability analysis to assess whether these vulnerabilities can propagate to the core logic of the target project, thereby reducing false positives.

1) Vulnerability Existence Analysis

Projects may reuse only parts of a TPL, while vulnerabilities lie outside the reused functions, leading to false positives. To address this, *VulSCA* verifies whether the vulnerable code is actually reused in the project.

As shown in Fig. 4, *VulSCA* retrieves the reused TPL community ID and queries the pre-built TPL vulnerability database to locate matching vulnerable communities. It then extracts the hash values and source code of the vulnerable functions in those communities, forming a candidate vulnerability library for the target project. Next, *VulSCA* applies *clone vulnerability detection* only to the functions within the reused communities, avoiding full-project scans and significantly improving efficiency.

By default, *VulSCA* employs *hash matching* for clone detection, comparing function hashes to check whether any reused function is identical to a known vulnerability. To handle more complex clone scenarios, *VulSCA* can also integrate with advanced detectors, such as *Vuddy* [4] for Type-II clones, and *Fire* [5] and *AntMan* [9] for Type-III clones. In our experiments (Table VII), we evaluated different configurations ($VulSCA_{Vuddy}$, $VulSCA_{Fire}$, and $VulSCA_{AntMan}$). The results show that replacing hash matching with advanced clone detectors substantially improves recall in vulnerability detection.

2) Vulnerability Reachability Analysis

VulSCA first identifies potential supply chain risks by locating known vulnerable functions within reused TPL communities. However, this does not indicate whether the vulnerabilities can actually affect the project, which may lead to false positives. Prior studies show that only reachable vulnerabilities are likely to pose real threats [23].

To address this, *VulSCA* incorporates reachability analysis, inspired by *Younis et al.* [8], to determine whether a path exists from the project's entry function to a vulnerable function. By operating at the community level rather than the function level, *VulSCA* avoids path explosion and improves efficiency. As shown in Fig. 4, the process consists of two stages: inter-community and intra-community reachability analysis.

Inter-Community Reachability Analysis. To mitigate the path explosion problem, we first conduct a coarse-grained analysis. Specifically, we check whether there exists a path from the community containing the project's entry function (entry community) to the community containing the vulnerable function (vulnerability community). If no path exists in the community-level call graph, no path can exist in the function-level call graph, making fine-grained analysis unnecessary.

To enable this, each semantic community is represented as a node, with edges established based on inter-community function calls, forming a community-level call graph. As illustrated in Fig. 4, the entry community is Com_1 (yellow node) and the vulnerability community is Com_3 (blue node). On average, each TPL in our library contains 3,522 functions grouped into 918 communities, a reduction that greatly decreases traversal overhead and effectively alleviates path explosion.

Intra-Community Reachability Analysis. If a path exists between the entry community and the vulnerability community in the community-level call graph, *VulSCA* proceeds with fine-grained reachability analysis inside the vulnerability community. This step examines whether vulnerable functions can be accessed from predecessor communities.

VulSCA verifies whether a vulnerable function is reachable from any predecessor community along the identified path.

TABLE III
DETAILS OF SOME METRICS IN THE SCA DATASETS

Metric	Abbr.	Definition in SCA Task	Definition in Vulnerability Detection Task
True Positive	TP	The number of correctly identified dependency relationships	The number of correctly detected reachable supply chain vulnerabilities
False Positive	FP	The number of incorrectly identified dependency relationships	The number of incorrectly detected reachable supply chain vulnerabilities
False Negative	FN	The number of missed dependency relationships	The number of missed reachable supply chain vulnerabilities
Area Under the Precision-Recall Curve	AUPRC	The area under the precision-recall curve	-
Recall	-	TP/(TP + FN)	
Precision	-	TP/(TP + FP)	
F1-score	-	2*Precision*Recall/(Precision + Recall)	

Specifically, we first identify the functions in the vulnerability community that are connected to its predecessors. These functions are treated as entry points to the vulnerability community (denoted F_{entry}, shown as a purple node in the "Intra-community Reachability Analysis" of Fig. 4). Subsequently, *VulSCA* checks whether a path exists from any F_{entry} to the vulnerable function F_{vul} within the community's call graph. If such a path exists, the vulnerability is considered reachable.

Even in fine-grained analysis, *VulSCA* restricts computation to the vulnerability community's call graph, which is a subgraph of the project call graph. This targeted scope further alleviates the path explosion problem often associated with function-level reachability analysis. In summary, through this two-stage reachability analysis, *VulSCA* determines whether a supply chain vulnerability poses a real threat to the target project, thereby reducing false positives.

IV. EXPERIMENT

In this section, our experiments focus on answering the following Research Questions (RQs):

- **RQ1**: What is the optimal parameter configuration for *VulSCA* to maximize its detection performance?
- **RQ2**: How does *VulSCA* perform in software composition analysis compared to state-of-the-art C/C++ SCA tools?
- **RQ3**: How does *VulSCA* perform in supply chain vulnerability detection compared to leading version-based and code-based methods?
- **RQ4**: What are the time and memory overheads of *VulSCA*, and how do they compare with existing tools in terms of efficiency and scalability?

A. Experimental Settings

1) Dataset

To support our experiments and analysis, we construct two related datasets. First, an SCA dataset is built by labeling high-quality C/C++ projects and their third-party dependencies, which serves to evaluate the SCA capability of *VulSCA*. On top of this dataset, we further annotate all reachable vulnerabilities to form a supply chain vulnerability dataset, which is used to evaluate the effectiveness of vulnerability detection.

SCA Dataset Construction. We first collect high-quality C/C++ repositories and manually label their TPL dependencies as ground truth. Inspired by prior work [3], we observe that such dependencies are typically stored in directories such as `/3rdparty`, `/deps`, and `/third_party`, or documented in files like `Readme.md` and `Copyright`.

- *Step 1: Selecting Projects.* To ensure quality, we select the 100 most-starred C/C++ projects on GitHub, as star count serves as a simple proxy for popularity and reliability. After filtering out unsuitable repositories (*e.g.,* header-only libraries), we retain 80 valid projects and identify 484 associated TPLs. These projects span diverse domains such as system tools, embedded software, network services, and scientific computing frameworks.
- *Step 2: Labeling Dependencies.* TPLs are identified through manual inspection of directory names and documentation, and validated by five experts with unanimous agreement. Reused functions confirm their presence in the target project, and we document the exact usage locations and version details for both projects and TPLs. The final SCA dataset contains 564 repositories (80 targets + 484 TPLs) and 862 labeled dependency relationships. The projects cover diverse domains including game engines, databases, and multimedia tools, with an average codebase size of 2.5 million lines of code.

Supply-Chain Vulnerabilities Dataset. We then identify and annotate reachable supply chain vulnerabilities in the 80 target projects (aligned with the SCA dataset) through the following two steps:

- *Step 1: Supply Chain Vulnerability Scanning.* We collect source files from typical TPL directories and use the *Fire* dataset (built on *PatchDB* [24]) as the vulnerability source. Five clone detection tools (*i.e., hash*, *Vuddy* [4], *Fire* [5], *V1Scan* [1], and *AntMan* [9]) are applied to identify Type-I to Type-III clone vulnerabilities. After deduplication, we obtain 203 unique vulnerabilities.
- *Step 2: Vulnerability Reachability Analysis.* Following [8], we analyze call graphs to determine whether each vulnerability is reachable from an entry function. Nine security engineers, divided into three groups, independently verify the results, and only vulnerabilities confirmed by all three groups are labeled as reachable. In total, 53 of the 203 vulnerabilities are confirmed as reachable and serve as ground truth in our vulnerability dataset.

2) Evaluation Metrics

Following prior Following prior works [25], [7], [5], [3], we adopt a set of widely used metrics to evaluate the effectiveness of *VulSCA*. As shown in Table III, these metrics are defined separately for two tasks: the SCA task and the vulnerability detection task. For SCA, the focus is on whether dependency relationships between projects and TPLs are correctly identified, while for vulnerability detection, the focus shifts to whether reachable supply chain vulnerabilities are correctly

recognized. Although the metrics share the same names (*e.g.,* TP, FP, FN, Precision, Recall, and F1-score), their definitions differ depending on the task. We also include *Area Under the Precision-Recall Curve* (AUPRC), which provides a more comprehensive assessment of the precision–recall trade-off in SCA evaluation.

3) Baselines

To evaluate *VulSCA*, we select six representative baselines for C/C++ projects: *CENTRIS* [7], *OSSFP* [3], *V1Scan* [1], *Vuddy* [4], *Fire* [5], and *AntMan* [9]. Among them, *CENTRIS* and *OSSFP* are SCA tools that can also be applied to supply chain vulnerability detection, while the others are vulnerability detection tools without direct SCA support. Accordingly, *CENTRIS* and *OSSFP* serve as baselines for the SCA task, and all six methods serve as baselines for the vulnerability detection task, since *CENTRIS* and *OSSFP* can determine vulnerability presence via their SCA results.

• ***CENTRIS*** [7]: it addresses nested clones in TPLs by deduplicating functions, keeping only the earliest version. It then measures project similarity by the proportion of identical functions to infer potential dependencies.

• ***OSSFP*** [3]: it reduces noise by filtering out duplicated, trivial, and widely shared functions, treating the remainder as core TPL functions. A project is considered dependent on a TPL if it contains any of these core functions.

• ***V1Scan*** [1]: it detects vulnerabilities at the version level by applying SCA (*CENTRIS* by default) and then checking if patch code changes appear in the TPL to reduce FPs. Since its SCA results rely on external tools, *V1Scan* is used only for vulnerability detection, not SCA evaluation.

• ***Vuddy*** [4]: it normalizes functions and computes MD5 hashes for matching. It supports Type-I and Type-II clones, effectively detecting clones with identical structures or renamed identifiers, but performs poorly on heavily modified code.

• ***Fire*** [5]: it extracts tokens from modified functions and applies taint-path analysis to track data flow, enabling the detection of structurally altered yet semantically similar clones. It offers moderate support for Type-III clones.

• ***AntMan*** [9]: it constructs normalized function call graphs and extracts inter-procedural code property clusters to generate fine-grained vulnerability signatures. It handles complex Type-III clones well and is particularly effective at identifying significantly mutated vulnerabilities.

Hyperparameter tuning is applied to all baselines that support threshold configuration, while *CENTRIS*, which does not allow threshold adjustment, is evaluated with its default settings. As noted in the *OSSFP* paper [3], the F1-score, as the harmonic mean of precision and recall, effectively captures the trade-off between the two. Accordingly, we report the configuration that yields the highest F1-score for each method to ensure fair comparison. The detailed parameter settings of the baselines are as follows:

CENTRIS deduplicates functions by retaining the earliest version and infers dependencies when projects share $\geq$10% identical functions. *OSSFP* filters out 50% of simple functions and excludes common ones based on a 0.2 threshold, treating the remainder as core TPL functions. *V1Scan* performs vulnerability detection at the version level, using *CENTRIS* for SCA results and checking whether patch code changes appear in TPLs. *Vuddy* normalizes functions and applies MD5 hashing to detect Type-I and Type-II clones, but struggles with heavily modified code. *Fire* extracts tokens and applies taint-path analysis to detect semantically similar clones, filtering out 70% of functions via token analysis and 60% of the remainder via AST filtering. *AntMan* builds normalized function call graphs and code property clusters to generate vulnerability signatures, concluding a vulnerability is reproduced when matching vulnerability clusters exceed 70% while matching fix clusters remain below 70%.

4) Implementation Details

We conducted the experiments on a standard server equipped with 128 GB of RAM, an 8-core Intel Xeon CPU, and an NVIDIA RTX A6000 GPU, running Ubuntu 20.04. *VulSCA* integrates several TPLs to support its core functions. It uses *ctags* [26] to extract functions, *re* [27] to normalize code, and *hashlib* [28] to generate unique function hashes. The call graph is constructed with *Doxygen* [15] and partitioned into communities using *Infomap* [29] and *NetworkX* [30]. Finally, *NetworkX* performs reachability analysis to assess vulnerability propagation.

B. RQ1: Parameter Selection

In this section, we examine the optimal configuration of key parameters in *VulSCA*, focusing on the choice of community detection algorithm and the community similarity threshold θ.

1) Parameter Sharing Across Tasks

VulSCA performs two interdependent tasks: TPL reuse identification and supply chain vulnerability detection. These tasks are executed sequentially, with vulnerability detection relying on the TPL communities identified in the first phase. To maintain consistency, *VulSCA* adopts a unified parameter configuration, specifically the community detection algorithm and the threshold θ, for both tasks. Adjusting these parameters not only influences the accuracy of TPL reuse identification but also directly affects the effectiveness of vulnerability detection.

The community detection algorithm determines how precisely code is partitioned into communities, which in turn shapes both dependency identification and vulnerability analysis. The similarity threshold θ controls whether a dependency is established between a target project and a TPL community: a higher threshold favors precision but risks missing valid dependencies (increasing FNs), whereas a lower threshold favors recall but may introduce spurious matches (increasing FPs). Errors in dependency identification inevitably propagate into the vulnerability detection stage, thereby affecting its overall accuracy.

2) Community Detection Algorithm Selection

Following *OSSFP*'s observation that SCA performance is best reflected by the overall F1-score, we select the optimal community detection algorithm for *VulSCA* based on its highest F1-score. Specifically, we evaluate four algorithms (*Louvain* [16], *Leiden* [18], *Infomap* [17], and *Greedy Modularity*

TABLE IV
VulSCA'S VULNERABILITY DETECTION PERFORMANCE AT DIFFERENT COMMUNITY SIMILARITY THRESHOLDS (θ)

θ	0.97	0.96	0.95	0.94	0.93	0.92	0.91	0.90	0.89	**0.88**	0.87	0.86	0.85	0.84
TP	7	15	19	20	20	20	20	21	21	**25**	25	25	25	25
FP	9	10	11	11	11	13	13	13	13	**13**	16	16	16	16
F1-score	0.23	0.42	0.50	0.52	0.52	0.51	0.51	0.53	0.53	**0.60**	0.57	0.57	0.57	0.57
Precision	0.44	0.60	0.63	0.65	0.65	0.61	0.61	0.62	0.62	**0.66**	0.61	0.61	0.61	0.61
Recall	0.15	0.33	0.41	0.43	0.43	0.43	0.43	0.46	0.46	**0.54**	0.54	0.54	0.54	0.54

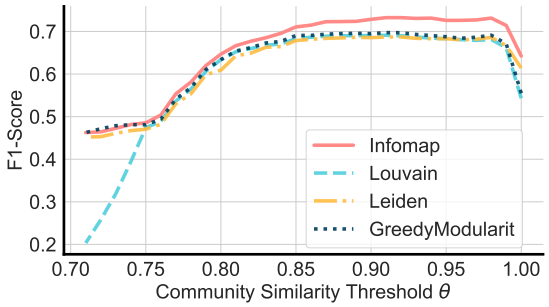

Fig. 5. SCA Results of *VulSCA* under Varying Community Detection Methods

[19]), to partition the call graphs and build the TPL feature repository. The target project's call graph is partitioned using the same algorithm. A similarity threshold $\theta \in [0.7, 1]$ is applied to determine whether a target community depends on a TPL community.

As shown in Fig. 5, *VulSCA* achieves the best SCA performance with *Infomap*, which is especially sensitive to community size and tends to cluster tightly related functions into smaller, semantically coherent communities. This granularity aligns well with SCA requirements, where clear semantics are essential for accurate dependency identification. In contrast, other algorithms often produce larger communities by grouping loosely related functions, introducing semantic noise that hinders precise matching. Therefore, given *Infomap*'s clear advantage in SCA, we adopt it as the default community detection algorithm in *VulSCA*.

3) Community Similarity Threshold Selection

Although *VulSCA* using the *Infomap* demonstrates strong and stable SCA performance across a wide threshold range ($\theta \in [0.83, 0.97]$), the threshold that achieves the highest F1-score in SCA does not necessarily lead to optimal performance in downstream vulnerability detection. This is because the F1-scores of the two tasks are not necessarily positively correlated. A high SCA F1-score may arise from high recall but low precision, introducing false-positive dependencies that cause the downstream detector to report non-existent vulnerabilities. Conversely, high precision but low recall may result in missing critical dependencies, leading to undetected vulnerabilities.

To address this, we focus on the stable interval $\theta \in [0.83, 0.97]$ and select the final threshold based on empirical results from the vulnerability detection task. Within this range, $\theta = 0.88$ achieves the best trade-off. It maintains accurate dependency identification while improving recall in vulnerability detection, without introducing excessive FPs. We therefore adopt $\theta = 0.88$ as the default threshold, following the same principle used by the baselines, where the parameter is selected based on the highest F1-score.

***Answer to RQ1**: VulSCA selects Infomap as the final community detection algorithm based on its performance in SCA, with the community similarity threshold θ set to 0.88 according to its vulnerability detection performance.*

C. RQ2: SCA Comparative Experiments

This subsection evaluates *VulSCA*'s detection performance for C/C++ SCA through comparative experiments. The experiments use the SCA dataset described in Section IV-A1 and compare *VulSCA* with *CENTRIS* [7] and *OSSFP* [3].

According to Table V, *VulSCA* outperforms both *CENTRIS* and *OSSFP* in overall detection capability (F1-Score), with improvements of 4% and 12%, respectively. In terms of precision, *VulSCA* achieves the highest performance by leveraging community-level similarity analysis, exceeding *CENTRIS* and *OSSFP* by over 19%. For recall, *VulSCA* surpasses *CENTRIS* but is slightly lower than *OSSFP*, due to its stricter criteria for determining dependency communities, which prioritize precision at the cost of partial recall. *VulSCA* also achieves a higher AUPRC than *OSSFP*, demonstrating superior overall performance across different parameter settings. Note that *CENTRIS* is excluded from the AUPRC comparison in Table V because its open-source implementation does not support parameter tuning.

1) Comparison with OSSFP

VulSCA achieves significantly higher precision than *OSSFP*. In practice, inconsistent version management in some TPLs can lead to inaccurate timestamps, causing *OSSFP* to misattribute core functions and generate FPs. To address this, *VulSCA* introduces a virtual node substitution mechanism: functions identified as non-original are replaced with virtual nodes, and TPL identification is performed based on community-level similarity. By jointly considering code text and call graph structure, this method tolerates partially misidentified virtual nodes and mitigates the limitations of timestamp-based approaches in handling nested dependencies. As a result, *VulSCA* offers improved precision and robustness in complex real-world settings.

As for recall, *OSSFP*, a function-level SCA approach, identifies dependencies whenever any core function from a

TPL appears in the target repository. Following the common tuning strategy adopted by all tools, which selects parameters to maximize the F1-score as described in the *OSSFP* paper [3], *VulSCA* applies stricter community-level dependency criteria to achieve higher precision, although this results in a slight reduction in recall.

TABLE V
COMPARISON OF SCA EFFECTIVENESS AND EFFICIENCY BETWEEN *CENTRIS*, *OSSFP*, AND *VulSCA*

SCA Method	TP	FP	Precision	Recall	F1-Score	AUPRC
CENTRIS	538	401	0.57	0.62	0.60	-
OSSFP	715	520	0.58	0.83	0.68	0.582
VulSCA	588	176	0.77	0.68	0.72	0.634

2) Comparison with CENTRIS

As shown in Table V, *CENTRIS* has 20% lower precision compared to *VulSCA*. *CENTRIS* detects dependencies by assessing the similarity between two files after excluding identical functions, based on the proportion of shared functions. However, due to the impact of chaotic version control on function-TPL assignments, this approach may result in files being misattributed to the wrong TPL, leading to FPs. In contrast, *VulSCA*, with its community-level dependency detection, is more robust to such issues, which minimizes their impact and results in higher precision.

Regarding recall, *CENTRIS*, as a file-level SCA method, encounters difficulties when a file contains functions from multiple TPLs, as the similarity between each TPL and the file may be low, hindering effective dependency detection. In contrast, *VulSCA* operates at the community-level, aiming to group functions from the same TPL into the same community, thereby strengthening their relationships. As a result, *VulSCA* significantly reduces false negatives.

3) VulSCA False Positive Analysis

VulSCA achieves the highest SCA precision; however, as shown in Table V, it still incurs 176 FPs. To investigate the root causes, we manually examine all FPs and attribute each to the responsible component and underlying technique within *VulSCA*. The results of this analysis appear in Table VI.

• **Input Dataset**: Incomplete TPL coverage in the dataset leads to 30 FPs, accounting for 17.04% of all cases. No experimental dataset comprehensively covers all open-source repositories. Consequently, when two projects depend on the same TPL that is missing from the dataset, the SCA method may incorrectly infer a dependency between them, producing FPs. This limitation affects *VulSCA* as well as all baselines. However, because all tools are evaluated on the same dataset, the impact remains consistent and the comparison remains fair.

• **Call Graph Construction**: At this stage, complex nested dependencies cause 94 FPs, accounting for 51.13% of all cases. These errors stem from poorly maintained open-source repositories or disorganized project structures, which generate abnormal function creation timestamps. As a result, *VulSCA* may misidentify the earliest defined function and incorrectly attribute the original function to the wrong TPL, thereby reporting false dependencies. Timestamp anomalies also affect *OSSFP* and *CENTRIS*; however, as discussed in the *Comparison with* OSSFP section, *VulSCA* introduces the virtual-node substitution mechanism, which helps mitigate this issue to some extent.

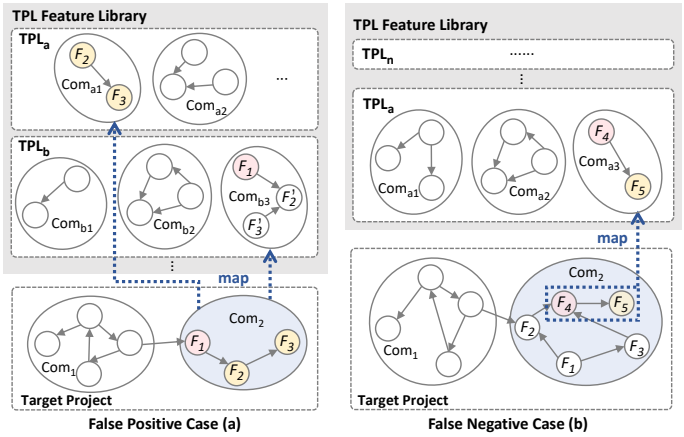

Fig. 6. FP and FN Cases in *VulSCA*'s SCA

• **Community Partitioning**: A total of 52 FPs, accounting for 29.54% of all cases, were caused by community-level granularity. Compared to function-level analysis, community-level analysis is coarser. *VulSCA* considers both function code and graph structure when evaluating community similarity, which may lead to false dependencies between communities with similar code and structure. For example, as shown in Fig. 6 (a), TPL_a originally creates functions F_2 and F_3, while TPL_b modifies them slightly to create F_2' and F_3' and introduces a new function F_1 that calls F_2' and F_3'. The target project directly reuses F_2 and F_3, creating a function F_1 to call them. Due to the similarities in function code and graph structure between community Com_2 of the target project and Com_{a1} of TPL_a, *VulSCA* mistakenly detects community Com_2 as dependent on Com_{b3}.

4) VulSCA False Negative Analysis

VulSCA results in a total of 274 FNs. Their root causes are identified through manual analysis, as outlined below.

• **Call Graph Construction**: In this module, the *Call Graph Extraction* and *Nested Dependency Elimination* steps account for 90 and 38 FNs, respectively.

The *Call Graph Extraction* step is limited by the capabilities of static analysis tools, which may miss certain function call relationships. As a result, *VulSCA* cannot accurately partition communities based on the incomplete call graph, leading to FNs. This issue accounts for 32.84% of all FNs.

The *Nested Dependency Elimination* step contributes 13.86% of the FNs. These errors arise from inaccurate timestamps, which affect all three SCA tools. When duplicate functions are incorrectly recorded as created before the originals, the dependency direction is reversed, causing the original repository to be mistakenly identified as dependent on the referencing one.

TABLE VI
FP AND FN DISTRIBUTION ACROSS DIFFERENT PHASES OF THE SCA TASKS IN *VulSCA*

Module	FP	FN	Root Cause
Input Dataset	30 (17.04%)	0	Incomplete TPL coverage in dataset
Call Graph Construction	0	90 (32.84%)	Inaccurate call graph extraction
	94 (51.13%)	38 (13.86%)	Complex nested dependency
Community Partitioning	52 (29.54%)[1]	146 (53.28%)[2]	Limitation of community granularity

[1] Detailed FP cases are visualized in Fig. 6 (a).
[2] Detailed FN cases are visualized in Fig. 6 (b).

• **Community Partitioning**: Community granularity produces 146 FNs, accounting for 53.28% of all FN cases. This typically occurs when a project reuses only a few functions from a TPL, which are insufficient to form a cohesive community in the call graph. As a result, the partitioned community contains a high proportion of functions authored by the target project, while the semantic contribution from the TPL is limited. Consequently, *VulSCA* fails to identify the dependency between the community and the original TPL.

For example, as shown in Fig. 6 (b), when the target project reuses only two functions from TPL_a, the majority of functions in the resulting community are written by the target project. This imbalance reduces the semantic presence of TPL_a in the community, which makes it difficult for *VulSCA* to detect the actual dependency.

***Answer to RQ2**: Compared with the baselines, VulSCA demonstrates superior performance in real-world SCA scenarios, achieving an F1-score of 0.72 and surpassing OSSFP and CENTRIS by 4–12%.*

D. RQ3: Vulnerability Detection Comparative Experiments

This subsection evaluates two aspects: (1) the impact of different clone vulnerability detection methods on *VulSCA*'s performance, and (2) its effectiveness compared with other supply chain vulnerability detection methods.

1) Effectiveness of VulSCA with Different Clone Vulnerability Detection Methods

To assess the influence of clone detection techniques and explore the flexibility of *VulSCA* to more complex clone types, we replace its default *Vulnerability Existence Analysis* algorithm (*i.e., hash*) with three advanced methods: *Vuddy* [4], *Fire* [5], and *AntMan* [9].

We then evaluate *VulSCA*'s performance in supply chain vulnerability detection based on the clone vulnerabilities identified in TPLs by each method. The rows labeled "*VulSCA*" in Table VII report its performance under different clone detection settings. Replacing the default hash-based approach with advanced tools such as *Vuddy*, *Fire*, and *AntMan* increases the number of vulnerabilities detected and demonstrates that *VulSCA* can adapt to different levels of clone complexity.

In terms of TPs, all alternative methods outperform $VulSCA_{hash}$. $VulSCA_{Fire}$ detects the most TPs (31), followed by $VulSCA_{AntMan}$ (28) and $VulSCA_{Vuddy}$ (27). *Fire* and *AntMan* surpass *Vuddy* by identifying Type-III clones. Although *AntMan* is designed to handle more complex clones through normalized call graphs and interprocedural attribute clustering, its TP count is slightly lower than that of $VulSCA_{Fire}$. This may be because Type-III clones are relatively rare in the dataset and some call relations are missed during graph construction due to tool limitations. Overall, all three methods improve detection performance compared with the default hash-based approach.

However, $VulSCA_{Fire}$ yields the highest number of FPs, primarily because *Fire* does not incorporate reachability analysis. Although it detects many vulnerabilities, a significant portion are not actually invoked by the target project. *Fire* originally reports 113 FPs, and while $VulSCA_{Fire}$ reduces this number to 25 through its own reachability analysis, some unreachable instances remain.

In summary, *VulSCA* exhibits strong extensibility by supporting the integration of advanced clone vulnerability detection tools. Tools such as *Vuddy*, *Fire*, and *AntMan* further enhance its effectiveness in supply chain vulnerability detection.

TABLE VII
COMPARISON OF VULNERABILITY DETECTION EFFECTIVENESS BETWEEN VULSCA AND OTHER DETECTION SOLUTIONS

Group	Method	FP	TP	F1	Precise	Recall
Version-based	*OSSFP*	560	22	0.07	0.04	0.42
	CENTRIS	310	22	0.11	0.07	0.42
Code-based	*Vuddy*	62	30	0.41	0.33	0.57
	Fire	113	34	0.34	0.23	0.64
	AntMan	75	30	0.38	0.29	0.57
Code-based Combined with Version-based	*OSSFP-Hash*	35	28	0.48	0.44	0.53
	OSSFP-Vuddy	41	28	0.46	0.41	0.53
	OSSFP-Fire	87	33	0.38	0.28	0.62
	OSSFP-AntMan	44	29	0.46	0.40	0.55
	V1Scan	46	17	0.29	0.27	0.32
VulSCA	$VulSCA_{hash}$ *(default)*	13	25	0.55	0.66	0.47
	$VulSCA_{Vuddy}$	13	27	0.58	0.68	0.51
	$VulSCA_{Fire}$	25	31	0.57	0.55	0.58
	$VulSCA_{AntMan}$	20	28	0.55	0.58	0.53

2) Comparison between VulSCA and Baselines for Supply Chain Vulnerability Detection

To assess the effectiveness of *VulSCA*, we compare it against several representative baselines for supply chain vulnerability detection. These include version-based SCA tools, code-based clone vulnerability detectors, and hybrid approaches that combine SCA with vulnerability detection techniques.

Comparison with Version-based Methods. The rows labeled "Version-based" in Table VII show the detection results when C/C++ SCA methods are directly applied to supply chain vulnerability detection. These methods assume that if a target

TABLE VIII
FP AND FN DISTRIBUTION ACROSS DIFFERENT PHASES OF THE VULNERABILITY DETECTION TASKS IN *VulSCA*.

Module	FN				FP[1]				Root Cause
	$VulSCA_{hash}$	$VulSCA_{Vuddy}$	$VulSCA_{Fire}$	$VulSCA_{AntMan}$	$VulSCA_{hash}$	$VulSCA_{Vuddy}$	$VulSCA_{Fire}$	$VulSCA_{AntMan}$	
TPL Reuse Identification	1 (3.57%)	1 (3.84%)	1 (4.54%)	1 (4%)	0	0	0	0	Missing TPL dependency Identification
Vulnerability Existence Analysis	14 (50%)	12 (46.15%)	8 (36.36%)	11 (44%)	0	0	0	0	Limited performance of clone detection
Vulnerability Reachability Analysis	13 (46.42%)	13 (50%)	13 (59.09%)	13 (52%)	13 (100%)	13 (100%)	25 (100%)	20 (100%)	FN: imprecise call graph construction FP: incomplete path checking

1 Detailed FP cases are visualized in Fig. 7.

project depends on a vulnerable TPL, the project is affected and should be flagged accordingly. Specifically, they identify the TPLs and versions used by the target project, query vulnerability databases for known issues in those versions, and report the results as supply chain vulnerabilities [1].

However, *VulSCA* significantly outperforms both *CENTRIS* and *OSSFP*, achieving 44–48% higher F1-scores. This improvement arises from two major limitations of version-based approaches: (1) they generate many FPs due to the lack of code-level validation, and (2) their recall is constrained by incomplete or inaccurate component–version mappings in vulnerability databases. As a result, these methods often miss real vulnerabilities while reporting numerous irrelevant ones. Overall, the findings indicate that direct application of existing version-based SCA techniques is inadequate for accurate supply chain vulnerability detection.

Comparison with Code-based Methods. The "Code-based" rows in Table VII show the results of directly applying clone-based vulnerability detection to supply chain scenarios. These methods search for code in the target project that resembles known vulnerable functions and assume that all such matches pose real risks.

When integrated with clone detection tools, *VulSCA* improves their original F1-scores by 17–23% and increases precision by 29–35%. This demonstrates that *VulSCA*'s reachability analysis substantially enhances the accuracy of vulnerability detection. Code-based methods rely solely on code similarity without verifying whether the matched vulnerable functions are actually invoked by the target project. This limitation arises from the high cost of function-level reachability analysis, which often suffers from path explosion and excessive time or memory consumption. *VulSCA* mitigates this problem by operating at the community level.

Comparison of Version-based Combined with Code-based Approaches. To fairly evaluate *VulSCA*, which integrates SCA with clone vulnerability detection, we compare it with two baselines that adopt similar integration strategies. The first is a set of enhanced *OSSFP* variants (*OSSFP-**), where the original *OSSFP* is augmented with different clone detection modules to enable function-level vulnerability identification. The second is *V1Scan*, which extends *CENTRIS* by incorporating patch-based validation for vulnerability detection.

While *CENTRIS* alone is a widely used SCA tool, it operates at the file level and lacks the granularity to determine which specific TPL functions are actually used in the target project.

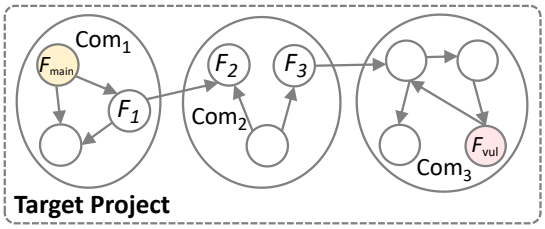

Fig. 7. FP Cases in *VulSCA*'s Vulnerability Detection

This limitation prevents it from directly supporting precise clone vulnerability detection. As a workaround, *V1Scan* augments *CENTRIS* with vulnerability-specific patch matching. For consistency and completeness, we re-implement *V1Scan* for comparison in our evaluation.

The results show that *VulSCA* consistently outperforms all these combined approaches, achieving F1-score improvements ranging from 7% to 19%. Notably, when paired with a more powerful clone detector such as *Fire*, *VulSCA* achieves even greater gains, surpassing *OSSFP-Fire* by 19% and *V1Scan* by 28%. In particular, *VulSCA* exhibits substantially higher precision than these approaches, which is primarily attributed to its reachability analysis that effectively filters out non-invoked vulnerabilities.

Reasons for *VulSCA*'s Higher Precision. *VulSCA*'s superiority in detection accuracy over other detection methods primarily stems from three key innovations:

- *Community-level Dependency Detection*: *VulSCA* employs community-level SCA detection, which not only provides reliable support for subsequent supply chain vulnerability detection but also successfully avoids the path explosion problem common in function-level path analysis.
- *Vulnerability Existence Analysis*: After completing SCA detection, *VulSCA* further integrates clone detection technology to filter out unreused third-party vulnerable functions in the project. Compared with traditional version-based schemes, this method can significantly reduce FPs.
- *Vulnerability Reachability Analysis*: After detecting vulnerability, *VulSCA* further verifies whether these are reachable from the project's entry point. This additional validation improves detection accuracy. In contrast, existing version-based and code-based methods lack this path-sensitive analysis. Even when combining the two, they cannot address the high overhead of reachability analysis, leading to more FPs.

TABLE IX
COMPARISON OF EFFICIENCY BETWEEN *VulSCA* AND OTHER DETECTION SOLUTIONS

Group	Method	Software Composition Analysis Time (s)		Vulnerability Detection Time (s)		Total Time (s)	Avg. Time (s)	Peak Memory Usage (GB)	Average Memory Usage (GB)
		Feature Extraction Time (s)	TPL Identification Time (s)	Detection Time (s)	Reachability Analysis Time (s)				
Version-based	*OSSFP*	6,382	12	1	-	6,395	80	10.2	4.5
	CENTRIS	6,588	16,579	1	-	23,168	290	11.9	5.2
Code-based	*Vuddy*	8,027	-	3,440	-	11,467	143	9.9	6.9
	Fire	2,464	-	9,856	-	12,320	154	10.1	7.8
	AntMan	19,553	-	13,035	-	32,588	407	7.9	6.6
Code-based Combined with Version-based	*OSSFP-Hash*	6,376	12	1	-	6,388	80	10.2	4.3
	OSSFP-Vuddy	6,762	12	166	-	6,939	87	10.2	4.5
	OSSFP-Fire	6,775	12	1,601	-	8,388	105	10.2	5.1
	OSSFP-AntMan	7,316	12	627	-	7,956	99	10.2	4.7
	V1Scan	6,588	16,579	2,385	-	25,544	319	11.9	4.7
VulSCA	$VulSCA_{Hash}$ *(default)*	7,011	46	1	182	7,240	91	16.2	7.2
	$VulSCA_{Vuddy}$	7,162	46	351	189	7,748	97	16.2	7.1
	$VulSCA_{Fire}$	7,375	46	1,455	310	9,187	115	16.2	7.3
	$VulSCA_{Antman}$	7,867	46	570	214	8,697	109	16.2	7.1

***VulSCA* False Positive Analysis.** Manual analysis shows that all FPs in *VulSCA* stem from limitations in its reachability analysis, as summarized in Table VIII. To improve efficiency, *VulSCA* performs reachability analysis in two stages: (1) inter-community analysis, from the project's entry community to the vulnerable community, and (2) intra-community analysis, within the vulnerable community itself, from its entry function to the vulnerable function.

However, it skips intra-community checks for intermediate communities along the path, except for the vulnerable community. As illustrated in Fig. 7, although F_2 in Com_2 is reachable from Com_1, there is no path from F_2 to the vulnerable function F_3 within Com_2. Since Com_2 is not the vulnerable community, its internal reachability is not analyzed, which leads to an FP.

***VulSCA* False Negative Analysis.** Manual analysis reveals that the FNs in *VulSCA* arise from three main phases: TPL reuse identification, vulnerability existence analysis, and vulnerability reachability analysis. Among these, the latter two account for the majority of cases, contributing 36% to 52% of all FNs, as shown in Table VIII.

• *TPL Reuse Identification:* This phase contributes a relatively small portion of FNs. Such cases occur when some dependency relationships are not successfully identified during the SCA phase. As a result, vulnerabilities in the reused TPLs cannot be detected, leading to FNs.

• *Vulnerability Existence Analysis:* The FNs in this phase vary depending on the clone detection algorithm used in *VulSCA*. They occur when the clone-based vulnerability detection tool fails to recognize the presence of a known vulnerable code snippet from a TPL in the target repository. Therefore, performance is closely tied to the capability of the underlying clone detection algorithm. As shown in Table VIII, clone detection tools with stronger detection capabilities, such as *Fire* (which supports Type-III clones), result in fewer FNs compared with simpler tools like *hash* (which supports only Type-I clones).

• *Vulnerability Reachability Analysis:* The FNs in this phase mainly result from inaccuracies in call graph construction, which lead to missing call edges. Consequently, when *VulSCA* performs reachability analysis, some call chains are incorrectly considered broken. These incomplete paths make vulnerabilities appear unreachable, ultimately causing FNs.

> ***Answer to RQ3****: VulSCA demonstrates strong flexibility in supply chain vulnerability detection, with its performance further enhanced by integration with advanced clone detection tools. On this task, it achieves 44–48% higher F1-scores than version-based methods and 17–23% higher than code-based methods.*

E. RQ4: Efficiency Analysis

This subsection analyzes the time and memory overhead of each tool to evaluate scalability and efficiency. The results are summarized in Table IX.

1) Time Overhead in the SCA Phase

The SCA phase typically consists of two steps: feature extraction and TPL identification, although their implementations differ across method groups. In terms of total SCA time, *VulSCA-Hash* runs significantly faster than *CENTRIS* and only slightly slower than *OSSFP*.

Version-based tools extract features after code preprocessing and perform matching during TPL identification. Code-based tools, which rely on clone detection, skip TPL identification and instead extract clone-relevant features. Hybrid methods (*i.e.,* version-based combined with code-based) and *VulSCA* follow a two-part process: extracting TPL features and generating clone vulnerability features based on the SCA results. Since hybrid methods share the same SCA process as version-based tools, we focus on *OSSFP*, *CENTRIS*, and *VulSCA* when comparing time overhead.

VulSCA's feature extraction is the most time-consuming step, as it involves call graph construction and community partitioning. In contrast, *OSSFP* computes function hashes directly, and *CENTRIS* applies locality-sensitive hashing (LSH). Although *VulSCA* introduces additional overhead, the cost remains acceptable given its improved accuracy.

For TPL identification, *OSSFP* is the fastest because it relies on simple hash matching. *CENTRIS* is the slowest,

since it compares every function pair between the project and TPLs using LSH, which incurs high computational cost. *VulSCA* improves efficiency by computing similarity at the community level rather than the function level, which drastically reduces the number of comparisons. Its lightweight community features and GPU-accelerated similarity further enhance performance, enabling it to complete identification in only 46 seconds.

2) Time Overhead in the Vulnerability Detection Phase

Vulnerability detection time consists of two components: detection time and reachability analysis time. Detection time refers to the time spent identifying supply chain vulnerabilities based on extracted features. Reachability analysis, which is supported only by *VulSCA*, determines whether a detected clone vulnerability is reachable from the project's entry point. Overall, version-based tools are the fastest, followed by hybrid methods (code-based combined with version-based) and *VulSCA*, while code-based tools are the slowest.

Version-based tools achieve the highest speed because they only query known vulnerabilities by matching TPL names and versions from the SCA results. However, without code-level semantic analysis, their detection F1-scores remain very low (0.07 and 0.11). Code-based methods perform exhaustive clone detection by comparing every project function against all known vulnerable functions, which introduces substantial overhead. In contrast, hybrid methods and *VulSCA* reduce this cost by narrowing the clone detection scope to the TPLs identified during the SCA phase. As a result, even with the additional reachability analysis step, *VulSCA* remains faster than pure code-based approaches.

Notably, under the same clone detection algorithm, *VulSCA* variants demonstrate higher efficiency than both *OSSFP*-based combinations and *V1Scan*. This improvement stems from *VulSCA*'s more accurate SCA results, which localize vulnerable dependencies more precisely and thereby reduce redundant clone matching. These findings underscore the importance of accurate SCA in minimizing vulnerability detection time.

3) Overall Time Overhead Analysis

VulSCA variants achieve better overall detection efficiency than all methods in the code-based group and perform comparably to *OSSFP*-based combinations, while delivering significantly higher detection accuracy.

The code-based group is noticeably slower because it lacks an SCA stage. Without SCA to narrow the scope of potential vulnerable TPLs, these methods must scan many irrelevant candidates, nearly doubling the total detection time. *VulSCA* incurs slightly more overhead than *OSSFP*-based combinations due to feature extraction and reachability analysis. Building the call graph and performing community partitioning introduce extra cost, but this is largely offset by the reduced comparison count during TPL identification. Each TPL contains an average of 3,522 functions, while community partitioning reduces this number to 918 on average, significantly lowering the number of comparisons. Reachability analysis adds little overhead, since it considers only inter-community paths and intra-community reachability within the vulnerable community.

On average, *VulSCA* requires about 10 seconds more per project than *OSSFP*-based combinations. However, this small increase yields about 10% higher detection accuracy, demonstrating that *VulSCA* achieves a favorable balance between efficiency and accuracy.

4) Memory Overhead Analysis

For tools that include TPL identification, peak memory usage occurs during this phase because TPL feature data must be loaded and compared. Among them, the *VulSCA*-* variants show the highest peak memory consumption. This is because *VulSCA* performs similarity computations on community-level features, which introduces additional memory overhead. As a result, both its peak and average memory usage are the highest, though still within an acceptable range. It is also worth noting that both *VulSCA* and *AntMan* utilize GPU acceleration. During execution, *VulSCA*'s peak GPU memory usage remains low at 3.1GB (less than 10% of total GPU memory), indicating minimal reliance on GPU resources.

***Answer to RQ4**: VulSCA maintains acceptable memory usage and comparable time efficiency while delivering higher detection accuracy, achieving a favorable balance between efficiency and accuracy.*

V. Discussion

A. Practicality of VulSCA

We apply *VulSCA* to detect supply chain vulnerabilities in 80 high-quality open-source projects. By combining community-level dependency detection with reachability analysis, *VulSCA* identifies 32 vulnerabilities with reachable call paths. To validate these findings, we perform small-scale directed fuzz testing using a state-of-the-art directed fuzzing tool [31]. This process confirms 18 vulnerabilities. For each confirmed case, we provide fix suggestions and submit detailed reports through GitHub Issues. To date, five project maintainers have acknowledged and merged our reports.

B. Call Graph Extraction

Although *VulSCA* employs state-of-the-art call graph construction tools, the generated graphs still suffer from incomplete call relationships. Moreover, even minor modifications in the call graph can lead to significant changes in community structures, thereby affecting the detection performance of *VulSCA*. In future work, we plan to design a more efficient and accurate call graph construction scheme to address the issue of incomplete call relationships and to enhance both the stability and effectiveness of *VulSCA*.

C. Similarity Metric Averaging

The current averaging strategy for combining structural- and code-level features in *VulSCA* is selected based on extensive empirical validation. Our analysis shows that structural and code features play distinct yet complementary roles in SCA scenarios. In earlier experiments, we applied independent similarity thresholds for structure (V_s) and code (V_c) features using grid search. However, this approach decreases the F1-score

(from 0.72 to 0.70) and introduces instability, as the decision boundaries become more sensitive to feature noise. In contrast, the averaging strategy demonstrates stronger robustness and better generalizability across diverse projects. Nevertheless, we acknowledge the asymmetric contributions of structural and code features. As future work, we plan to investigate adaptive weighting mechanisms to dynamically integrate the two feature types and further enhance detection performance.

D. Nested Dependency Handling

Existing methods typically rely on timestamps to resolve nested dependencies. However, in practice, issues such as repository migration, missing timestamps, or inconsistencies may cause the timestamp of a function's source repository to appear later than that of its reused repository. This reversal can mislead TPL reuse identification and result in FPs. To mitigate this issue, *VulSCA* introduces a virtual node substitution mechanism to enhance nested dependency handling. Specifically, functions identified as non-original within a community are replaced with virtual nodes, and SCA is then performed at the community level. This approach reduces the impact of inaccurate timestamps for individual functions and improves the SCA's robustness.

E. Dataset Selection Bias

Since the *OSSFP* dataset is not publicly available, we reproduced its construction process and further annotated reachable vulnerabilities associated with TPLs, which are essential for supply chain vulnerability detection. This additional annotation effort limited the dataset size but made it more suitable for evaluating SCA tools in the vulnerability detection task. In contrast, the original *OSSFP* dataset lacks vulnerability information and is thus not applicable to this task. As noted in Section IV.C, 30 FPs were caused by incomplete TPL coverage in our dataset. Nevertheless, all methods were evaluated on the same dataset, ensuring fair and consistent comparison. We plan to expand the dataset in future work to include more real-world projects and further assess the generalizability of *VulSCA*.

VI. Related Work

A. Software Composition Analysis

Developers increasingly clone TPL code to improve development efficiency. Studies report that 10–30% of code across repositories consists of cloned segments [32], [33]. However, code reuse also introduces security risks: even widely adopted TPLs such as *Log4j* contain numerous vulnerabilities that can propagate to many Java codebases through dependencies [34], [35]. *Zerouali* [36] highlights that clarifying dependency relationships can mitigate risks, while *Alfadel* [37] finds that excessively long vulnerability fix cycles exacerbate them. These findings underscore the need for SCA tools to manage TPL dependencies and address security risks.

For languages with official package managers such as Java and Python, SCA tools [38], [39], [40], [41] detect dependencies by parsing dependency files, and their effectiveness has been validated in empirical studies [42], [43], [44]. In contrast, C/C++ lacks official package management support, so SCA relies primarily on code clone detection techniques [45], [46], [47], which face notable limitations. For example, *OSSPolice* [2] uses directory structures as TPL signatures, but structural changes often lead to false negatives. *SourcererCC* [6] improves robustness but struggles with nested clones across TPLs. *CENTRIS* [7] addresses nested clones but suffers from prohibitively high time complexity. *OSSFP* [3] extracts core functions as TPL signatures, yet the omission of vulnerability-related functions in core selection limits its effectiveness for supply chain vulnerability detection.

B. Clone Vulnerability Detection and Reachability Analysis

Cloning TPL code into projects can also propagate vulnerabilities from TPLs to dependent projects. Code clone detection techniques [48], [49], [50], [51], [52], [53] have been widely applied to identify clones, and some studies [54], [4], [55], [25], [1] extend these approaches to detect cloned vulnerabilities. While effective, clone detection typically requires full repository scans to identify N-day vulnerabilities, making it difficult to respond promptly after vulnerability disclosures. Integrating SCA to narrow the scan scope can significantly improve efficiency and enable faster mitigation of security threats [56].

Most existing methods emphasize code similarity while overlooking whether detected vulnerabilities are actually reachable, which limits their practical applicability. Reachability analysis addresses this issue by evaluating whether vulnerabilities can propagate to execution paths, thereby reducing false positives. Several studies [57], [58], [59], [60], [23] have explored upstream vulnerability reachability. For example, symbolic execution has been used to analyze bug reachability in Python scientific libraries, though it relies heavily on conditions extracted from bug reports [58]. In the Java ecosystem, static and dynamic analyses have been applied to improve reachability assessment and enhance tools such as *Evosuite* [61]. However, these efforts are often ecosystem-specific and limited in scope. Research on vulnerability reachability in C/C++ remains scarce, highlighting the need for further investigation in this critical area.

VII. Conclusion

This paper tackles the challenge of C/C++ software supply chain vulnerability analysis by bridging the long-standing gap between SCA and vulnerability detection. We present *VulSCA*, the first community-level SCA framework that simultaneously supports dependency identification, vulnerability detection, and reachability analysis. By modeling call graphs as social networks, *VulSCA* extracts semantic communities to achieve more accurate dependency identification. Experimental results show that *VulSCA* outperforms traditional SCA tools such as *CENTRIS* and *OSSFP*, achieving 4–12% higher F1-scores in dependency detection. For supply chain vulnerability detection, it delivers even greater benefits, improving F1-scores by 17–48% compared to conventional methods.

ACKNOWLEDGEMENTS

We would thank the anonymous reviewers for their insightful comments to improve the quality of the paper. This work is supported by the Program of National Natural Science Foundation of China under Grant No. 62172168.

REFERENCES

[1] Seunghoon Woo, Eunjin Choi, Heejo Lee, and Hakjoo Oh. V1scan: Discovering 1-day vulnerabilities in reused c/c++ open-source software components using code classification techniques. In *32nd USENIX Security Symposium*, pages 6541–6556, 2023.

[2] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. Identifying open-source license violation and 1-day security risk at large scale. In *Proceedings of the 24th ACM SIGSAC Conference on computer and communications security*, pages 2169–2185, 2017.

[3] Jiahui Wu, Zhengzi Xu, Wei Tang, Lyuye Zhang, Yueming Wu, Chengyue Liu, Kairan Sun, Lida Zhao, and Yang Liu. Ossfp: Precise and scalable c/c++ third-party library detection using fingerprinting functions. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering*, pages 270–282, 2023.

[4] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *Proceedings of the 38th IEEE Symposium on Security and Privacy*, pages 595–614, 2017.

[5] Siyue Feng, Yueming Wu, Wenjie Xue, Sikui Pan, Deqing Zou, Yang Liu, and Hai Jin. Fire: Combining multi-stage filtering with taint analysis for scalable recurring vulnerability detection. In *Proceedings of the 33rd USENIX Security Symposium*, pages 1867–1884, 2024.

[6] Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K Roy, and Cristina V Lopes. Sourcerercc: Scaling code clone detection to big-code. In *Proceedings of the 38th international conference on software engineering*, pages 1157–1168, 2016.

[7] Seunghoon Woo, Sunghan Park, Seulbae Kim, Heejo Lee, and Hakjoo Oh. Centris: A precise and scalable approach for identifying modified open-source software reuse. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering*, pages 860–872, 2021.

[8] Awad A Younis, Yashwant K Malaiya, and Indrajit Ray. Using attack surface entry points and reachability analysis to assess the risk of software vulnerability exploitability. In *Proceedings of the 15th IEEE International Symposium on High-Assurance Systems Engineering*, pages 1–8, 2014.

[9] Yiheng Cao, Susheng Wu, Ruisi Wang, Bihuan Chen, Yiheng Huang, Chenhao Lu, Zhuotong Zhou, and Xin Peng. Recurring vulnerability detection: How far are we? *Proceedings of the ACM on Software Engineering*, 2(ISSTA):573–595, 2025.

[10] NGINX. https://github.com/nginx/nginx., 2025.

[11] Di Jin, Zhizhi Yu, Pengfei Jiao, Shirui Pan, Dongxiao He, Jia Wu, Philip S Yu, and Weixiong Zhang. A survey of community detection approaches: From statistical modeling to deep learning. *IEEE Transactions on Knowledge and Data Engineering*, 35(2):1149–1170, 2021.

[12] Splayer. https://github.com/tomasen/splayer, 2017.

[13] Openssl. https://www.openssl.org/, 2021.

[14] Dong Qiu, Bixin Li, and Hareton Leung. Understanding the api usage in java. *Information and software technology*, 73:81–100, 2016.

[15] Doxygen. https://www.doxygen.nl, 2025.

[16] Vincent D Blondel, Jean-Loup Guillaume, Renaud Lambiotte, and Etienne Lefebvre. Fast unfolding of communities in large networks. *Journal of statistical mechanics: theory and experiment*, 2008(10):P10008, 2008.

[17] Martin Rosvall and Carl T Bergstrom. Maps of random walks on complex networks reveal community structure. *Proceedings of the national academy of sciences*, 105(4):1118–1123, 2008.

[18] Vincent A Traag, Ludo Waltman, and Nees Jan Van Eck. From louvain to leiden: guaranteeing well-connected communities. *Scientific reports*, 9(1):1–12, 2019.

[19] Aaron Clauset, Mark EJ Newman, and Cristopher Moore. Finding community structure in very large networks. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics*, 70(6):066111, 2004.

[20] Haojun Zhao, Yueming Wu, Deqing Zou, and Hai Jin. An empirical study on android malware characterization by social network analysis. *IEEE Transactions on Reliability*, 73(1):757–770, 2023.

[21] Luai Al Shalabi, Zyad Shaaban, and Basel Kasasbeh. Data mining: A preprocessing engine. *Journal of Computer Science*, 2(9):735–739, 2006.

[22] National vulnerability database. https://nvd.nist.gov, 2021.

[23] Yulun Wu, Zeliang Yu, Ming Wen, Qiang Li, Deqing Zou, and Hai Jin. Understanding the threats of upstream vulnerabilities to downstream projects in the maven ecosystem. In *Proceedings of the 45th IEEE/ACM International Conference on Software Engineering*, pages 1046–1058, 2023.

[24] Xinda Wang, Shu Wang, Pengbin Feng, Kun Sun, and Sushil Jajodia. Patchdb: A large-scale security patch dataset. In *Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 149–160, 2021.

[25] Yang Xiao, Bihuan Chen, Chendong Yu, Zhengzi Xu, Zimu Yuan, Feng Li, Binghong Liu, Yang Liu, Wei Huo, Wei Zou, et al. Mvp: Detecting vulnerabilities using patch-enhanced vulnerability signatures. In *Proceedings of the 29th USENIX Security Symposium*, pages 1165–1182, 2020.

[26] ctags. https://github.com/universal-ctags/ctags, 2025.

[27] re. https://docs.python.org/3/library/re.html, 2025.

[28] hashlib. https://docs.python.org/3/library/hashlib.html, 2025.

[29] The MapEquation software package. https://mapequation.org, 2025.

[30] Software for complex networks (networkx). http://networkx.github.io, 2021.

[31] Penghui Li, Wei Meng, and Chao Zhang. Sdfuzz: Target states driven directed fuzzing. In *Proceedings of the 33rd USENIX Security Symposium*, pages 2441–2457, 2024.

[32] Mohammad Gharehyazie, Baishakhi Ray, Mehdi Keshani, Masoumeh Soleimani Zavosht, Abbas Heydarnoori, and Vladimir Filkov. Cross-project code clones in github. *Empirical Software Engineering*, 24:1538–1573, 2019.

[33] Rabe Abdalkareem, Emad Shihab, and Juergen Rilling. On code reuse from stackoverflow: An exploratory study on android apps. *Information and Software Technology*, 88:148–158, 2017.

[34] Saikat Chakraborty, Rahul Krishna, Yangruibo Ding, and Baishakhi Ray. Deep learning based vulnerability detection: Are we there yet? *IEEE Transactions on Software Engineering*, 48(9):3280–3296, 2021.

[35] Apache log4j2. https://github.com/apache/logging-log4j2., 2022.

[36] Ahmed Zerouali, Tom Mens, Alexandre Decan, and Coen De Roover. On the impact of security vulnerabilities in the npm and rubygems dependency networks. *Empirical Software Engineering*, 27(5):107, 2022.

[37] Mahmoud Alfadel, Diego Elias Costa, and Emad Shihab. Empirical analysis of security vulnerabilities in python packages. *Empirical Software Engineering*, 28(3):59, 2023.

[38] Eclipse steady. https://github.com/eclipse/steady/, 2022.

[39] Owasp dependency-check project - owasp. https://owasp.org/www-project-dependency-check/, 2021.

[40] About alerts for vulnerable dependencies - github docs. https://docs.github.com/en/code-security/supply-chain-security/managing-vulnerabilities-in-your-projects-dependencies/about-alerts-for-vulnerable-dependencies, 2023.

[41] Ochrona. https://ochrona.dev, 2021.

[42] Andreas Dann, Henrik Plate, Ben Hermann, Serena Elisa Ponta, and Eric Bodden. Identifying challenges for oss vulnerability scanners-a study & test suite. *IEEE Transactions on Software Engineering*, 48(9):3613–3625, 2021.

[43] Nasif Imtiaz, Seaver Thorn, and Laurie Williams. A comparative study of vulnerability reporting by software composition analysis tools. In *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–11, 2021.

[44] Lida Zhao, Sen Chen, Zhengzi Xu, Chengwei Liu, Lyuye Zhang, Jiahui Wu, Jun Sun, and Yang Liu. Software composition analysis for vulnerability detection: An empirical study on java projects. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 960–972, 2023.

[45] Abdullah Sheneamer and Jugal Kalita. A survey of software clone detection techniques. *International Journal of Computer Applications*, 137(10):1–21, 2016.

[46] Jingdong Wang, Heng Tao Shen, Jingkuan Song, and Jianqiu Ji. Hashing for similarity search: A survey. *arXiv preprint arXiv:1408.2927*, 2014.

[47] Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of computing TR*, 541(115):64–68, 2007.

[48] Siyue Feng, Wenqi Suo, Yueming Wu, Deqing Zou, Yang Liu, and Hai Jin. Machine learning is all you need: A simple token-based approach for effective code clone detection. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024.

[49] Tiancheng Hu, Zijing Xu, Yilin Fang, Yueming Wu, Bin Yuan, Deqing Zou, and Hai Jin. Fine-grained code clone detection with block-based splitting of abstract syntax tree. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 89–100, 2023.

[50] Yutao Hu, Deqing Zou, Junru Peng, Yueming Wu, Junjie Shan, and Hai Jin. Treecen: Building tree graph for scalable semantic code clone detection. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–12, 2022.

[51] Yuekun Wang, Yuhang Ye, Yueming Wu, Weiwei Zhang, Yinxing Xue, and Yang Liu. Comparison and evaluation of clone detection techniques with different code representations. In *Proceedings of the 45th International Conference on Software Engineering*, pages 332–344, 2023.

[52] Yueming Wu, Siyue Feng, Deqing Zou, and Hai Jin. Detecting semantic code clones by building ast-based markov chains model. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pages 1–13, 2022.

[53] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st IEEE/ACM International Conference on Software Engineering*, pages 783–794, 2019.

[54] Jiyong Jang, Abeer Agrawal, and David Brumley. Redebug: Finding unpatched code clones in entire os distributions. In *Proceedings of the 33rd IEEE Symposium on Security and Privacy*, pages 48–62, 2012.

[55] Benjamin Bowman and H Howie Huang. Vgraph: A robust vulnerable code clone detection system using code property triplets. In *Proceedings of the 5th IEEE European Symposium on Security and Privacy*, pages 53–69, 2020.

[56] Shangzhi Xu, Jialiang Dong, Weiting Cai, Juanru Li, Arash Shaghaghi, Nan Sun, and Siqi Ma. Enhancing security in third-party library reuse–comprehensive detection of 1-day vulnerability through code patch analysis. In *Proceedings of the 32nd Annual Network and Distributed System Security Symposium*, 2025.

[57] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vuln4real: A methodology for counting actually vulnerable dependencies. *IEEE Transactions on Software Engineering*, 48(5):1592–1609, 2020.

[58] Wanwangying Ma, Lin Chen, Xiangyu Zhang, Yang Feng, Zhaogui Xu, Zhifei Chen, Yuming Zhou, and Baowen Xu. Impact analysis of cross-project bugs on software ecosystems. In *Proceedings of the 42nd ACM/IEEE International Conference on Software Engineering*, pages 100–111, 2020.

[59] Henrik Plate, Serena Elisa Ponta, and Antonino Sabetta. Impact assessment for vulnerabilities in open-source software libraries. In *Proceedings of the 31st IEEE International Conference on Software Maintenance and Evolution*, pages 411–420, 2015.

[60] Ivan Pashchenko, Henrik Plate, Serena Elisa Ponta, Antonino Sabetta, and Fabio Massacci. Vulnerable open source dependencies: Counting those that matter. In *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*, pages 1–10, 2018.

[61] Emanuele Iannone, Dario Di Nucci, Antonino Sabetta, and Andrea De Lucia. Toward automated exploit generation for known vulnerabilities in open-source libraries. In *Proceedings of the 29th International Conference on Program Comprehension*, pages 396–400, 2021.