

Article

LLM-Enhanced Evolutionary Dependency Graphs: A System Approach to Precise Vulnerability Propagation Tracking

Yuanjun Gao ^{1,2}, Hongzhou Wu ², Yujia Luo ², Peichang Shi ², Wenhao Shao ³ and Jiuxin Cao ^{1,*} ¹ School of Cyber Science and Engineering, Southeast University, Nanjing 210096, China² KylinSoft Corporation, Changsha 410005, China; wuhongzhou01@kylinos.cn (H.W.); luoyujia@kylinos.cn (Y.L.); shipeichang@kylinos.cn (P.S.)³ School of computer electronics and information, Guangxi University, Nanning 530004, China

* Correspondence: jx.cao@seu.edu.cn

Abstract

With the widespread adoption of open-source software (OSS), software supply chain attacks have become increasingly sophisticated, posing severe threats to digital infrastructure. Traditional Software Composition Analysis (SCA) tools and Software Bill of Materials (SBOM) systems face critical limitations, including disconnection between static metadata and dynamic code evolution, inability to detect silent fixes and backported patches, and LLM-induced hallucinations in vulnerability detection. To address these challenges, this paper proposes the LLM-Enhanced Component Dependency Evolution Graph (CDEG) framework. The system constructs a time-aware heterogeneous graph by parsing SBOMs to establish static dependency edges and linking version lineages to model temporal evolution. Within this framework, we map code patches using a hybrid representation that fuses structural features extracted by Tree-sitter with semantic embeddings derived from a fine-tuned LLM, effectively distinguishing backported patches from code refactoring. Experimental results on the SC-Vuln-Real dataset demonstrate that our framework achieves 88.4% recall for backported patch detection and achieves an F1-Score of 93.8% in container image scanning.

Keywords: software supply chain security; vulnerability propagation; large language model (LLM); dependency graph; semantic analysis

MSC: 94A62; 68R10; 68T50



Academic Editors: Tao Zhang, Xiangyun Tang, Jiacheng Wang, Chuan Zhang and Jiqiang Liu

Received: 26 May 2026

Revised: 10 July 2026

Accepted: 15 July 2026

Published: 17 July 2026

Copyright: © 2026 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

1. Introduction

In modern software engineering paradigms, open source software (OSS) has become the cornerstone of digital infrastructure. Statistics show that over 90% of the code in modern applications consists of third-party open-source components [1]. However, this high degree of dependency makes the software supply chain a prime target for attackers. From the 2020 SolarWinds (2020.2 RC1) attack, to the 2021 Log4Shell (2.14.1) vulnerability, and the 2024 XZ Utils backdoor incident (CVE-2024-3094), the complexity and stealth of supply chain attacks are growing exponentially [2]. In particular, the XZ Utils incident indicates that attackers have begun to inject malicious logic into widely trusted foundational libraries through long-term social engineering and complex build script tampering—an attack method that completely bypasses traditional code signature-based defense systems.

Faced with this severe security situation, governments and standard organizations worldwide have vigorously promoted Software Bill of Materials (SBOM) systems (e.g.,

U.S. Executive Order 14028), aiming to manage risks through transparent component dependency relationships [3]. While SBOM provides a static snapshot of software components, this study argues that relying solely on SBOM and traditional Software Composition Analysis (SCA) tools is insufficient to address current threats, primarily due to three core challenges: As summarized in Figure 1, metadata-based SCA may produce false positives in backported-patch scenarios and false negatives in silent-fix scenarios, whereas the proposed LLM-Enhanced CDEG framework uses patch-level semantic evidence and dependency-evolution context to trace vulnerability propagation beyond version matching.

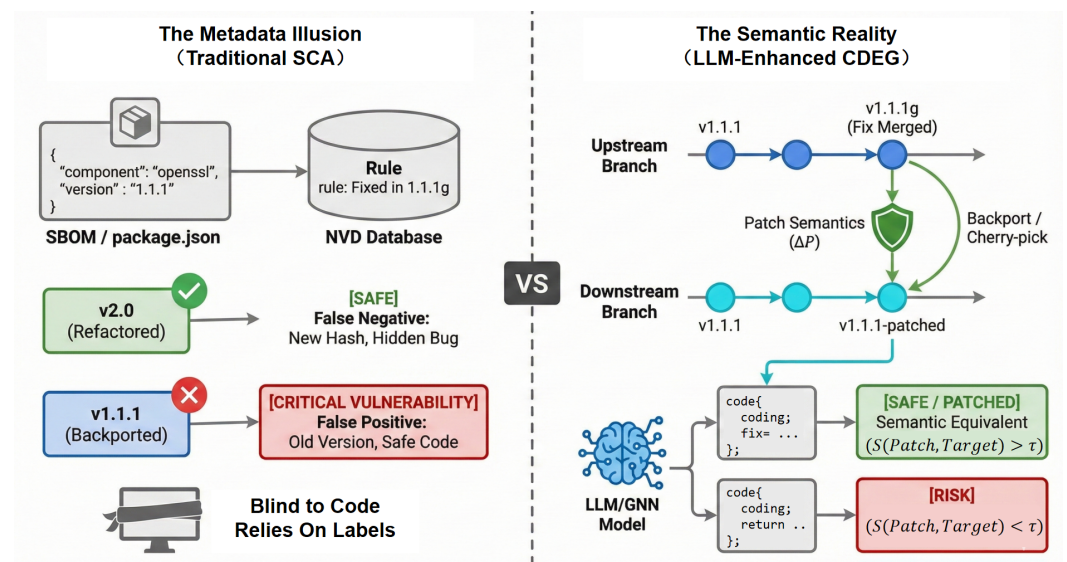


Figure 1. Split-screen comparison. **(Left)** Traditional SCA failing on “Backported Patch” (False Positive) and “Silent Fix” (False Negative) due to version metadata reliance. **(Right)** Our LLM-enhanced CDEG system successfully identifies the semantic equivalence of the patch despite version mismatches, utilizing a knowledge graph to trace the fix.

1.1. The SBOM Paradox: Disconnection Between Static Metadata and Dynamic Code Evolution

The core logic of traditional SCA tools is “version number matching”—determining risks by comparing version numbers in files such as `package.json` or `pom.xml` with CVE databases. However, in enterprise-level development and Linux distribution maintenance, the strategy of “version freezing with patch backporting” is prevalent [4]. To ensure stability, maintainers often merge security patches independently into older versions (e.g., `openssl-1.1.1`) without upgrading the major version number. This causes SCA tools to generate a large number of **False Positives** when version numbers remain unchanged after security backporting. Although modern SCA workflows automate SBOM generation, component identification, package-version extraction, and CVE database matching, security teams still need manual triage in several cases. These include verifying whether a reported vulnerability has already been fixed by a backported patch, checking whether a silent fix has been introduced without a public advisory, assessing whether distribution-specific modifications invalidate version-based conclusions, and determining whether the vulnerable function is actually reachable in downstream products or container images. Therefore, the limitation of traditional SCA in this scenario is not the lack of automation in component matching, but the lack of code-level and evolution-aware evidence for resolving ambiguous vulnerability states caused by backporting, silent fixes, and downstream customization.

1.2. Silent Propagation & Fixes

Studies have shown that a large number of security fixes are implemented in a “silent” manner—developers fix vulnerabilities without issuing CVEs or security announcements

to avoid attracting attackers' attention [5,6]. Traditional vulnerability scanners cannot detect these unpublicized fixes, leading to **False Negatives**. Additionally, code refactoring and custom modifications by downstream vendors alter code hash fingerprints, rendering hash-based detection techniques (e.g., SimHash) ineffective [7].

1.3. LLM Hallucinations and Context Deficiency in Security Domains

Although Large Language Models (LLMs) have demonstrated strong potential in code understanding and vulnerability detection [8], directly applying general-purpose LLMs to supply-chain vulnerability analysis remains unreliable. When models are asked to judge isolated code snippets without project-level evidence, dependency reachability, or patch history, they may produce unsupported or overconfident vulnerability judgments. Moreover, snippet-level LLM prompting cannot determine whether the analyzed code is actually invoked in downstream products or trace how vulnerable logic evolves across version trees [9]. Therefore, pure LLM-based analysis is insufficient for vulnerability propagation tracking and must be combined with structural code evidence and dependency-evolution context.

To address the aforementioned issues, this study proposes a novel system architecture integrating LLM semantic understanding and Evolutionary Dependency Graph (LLM-Enhanced CDEG). Instead of relying on fragile version numbers or hash matching, the system tracks the essence of vulnerabilities through semantic fingerprints. We use a fine-tuned code LLM to extract semantic features of patches—even if the code has been refactored or backported, the model can still identify the equivalence of its core logic. Furthermore, we map this semantic information into a Component Dependency Evolution Graph (CDEG), a multi-dimensional heterogeneous graph that includes time (version evolution), space (component dependency), and semantics (vulnerability propagation). By executing graph propagation algorithms on CDEG, the system can automatically infer the **True Impact Domain** of vulnerabilities across the entire supply chain. Rather than treating code representation, semantic matching, and graph traversal as separate pipeline stages, the proposed CDEG couples patch-level semantic evidence with dependency and version-evolution reasoning, enabling vulnerability propagation tracking across components and versions under backported-patch and silent-fix scenarios. The main contributions of this paper are as follows:

- We propose a novel structure-semantic hybrid patch representation that integrates data-flow-aware structural features with fine-tuned LLM-based patch-delta semantics. This representation is designed to identify semantically equivalent vulnerability fixes under backporting, code refactoring, and downstream customization scenarios.
- We design an LLM-enhanced Component Dependency Evolution Graph (CDEG) that extends conventional dependency and version evolution graphs by introducing semantic propagation edges derived from patch-level evidence. This enables vulnerability tracking across components and versions at the software ecosystem level, rather than relying only on version metadata or isolated code similarity.
- We conduct extensive evaluations on the SC-Vuln-Real dataset, which includes 18,156 real patches and 297 container images. Experimental results show that our system achieves 88.4% recall in detecting backported patches and reduces the false positive rate of container image scanning to 4.8%.

2. Background

2.1. SBOM and Software Supply Chain Security

With the frequent occurrence of major supply chain security incidents such as SolarWinds and Log4j, SBOM (Software Bill of Materials) has gradually become a core infrastruc-

ture for improving software supply chain transparency [10]. SBOM standards (e.g., SPDX, CycloneDX, SWID) are widely used to describe components, versions, build information, and dependency relationships contained in software systems, and are regarded as basic data for software supply chain security governance [11].

However, existing SBOM-related research still mainly focuses on automatic generation and parsing of SBOM, as well as component-level vulnerability identification methods based on SBOM [12]. For example, technologies for automatically generating SBOM in build systems are used to reduce manual maintenance costs, while research on recovering dependency relationships from binaries addresses scenarios where source code is unavailable [13]. In addition, traditional Software Composition Analysis (SCA) tools (e.g., OWASP Dependency-Check) parse component identifiers in SBOM and match them with vulnerability databases (e.g., NVD) to identify whether the system contains known vulnerabilities [14,15].

Although SBOM provides component-level visibility, its inherent characteristics make it difficult to support in-depth vulnerability governance [9]. First, SBOM itself does not contain source code-level semantic information, nor can it describe the evolutionary process within versions, making it difficult to distinguish whether patches are actually merged. Second, many enterprises implement localized modifications to third-party components, so version numbers no longer accurately reflect patch status. Studies have pointed out that this metadata-based matching mechanism is prone to generating a large number of false positives when dealing with the “backporting” strategy of distributions such as Debian or RedHat [16]. Finally, due to SBOM’s lack of ability to characterize code logic, when vulnerabilities propagate downstream through dependency chains, SBOM cannot track the actual propagation path. Therefore, although SBOM is an important foundation for supply chain security governance, it still has key deficiencies in vulnerability propagation analysis and impact domain assessment.

2.2. Code Semantic Analysis and Patch Understanding

To make up for the limitations of traditional text-based diff methods (e.g., SimHash algorithm) in handling code refactoring, semantic deformation, or API replacement, code semantic understanding and representation technologies have developed rapidly in recent years [17]. Structured analysis methods such as Abstract Syntax Trees (AST), Control Flow Graphs (CFG), and Data Flow Graphs (DFG) can express the syntactic and control features of programs, providing important information for patch analysis and vulnerability detection [17–19]. However, these structural features are usually sensitive to code refactoring and structural transformations, and are prone to misjudgment in cross-version and cross-branch comparison tasks.

With the application of deep learning technologies in code semantic modeling, pre-trained code models have provided new possibilities for cross-version semantic matching. CodeBERT proposed by Microsoft [20,21] first realized the alignment of natural language and code semantics through bimodal pre-training; subsequently, GraphCodeBERT [22] embeds data-flow features into pre-trained code representations to capture structural information beyond token similarity, which is important for robust code representation and semantic matching. However, it cannot identify equivalent vulnerability fixes in downstream components or model patch evidence propagation across dependencies and versions. Instead, our method integrates structural cues into a hybrid patch representation and links patch semantics to CDEG-based ecosystem-wide propagation analysis. The latest CodeT5 model [23] improves robustness in common obfuscation scenarios such as variable renaming through identifier-awareness tasks. By pre-training on a large amount of source code, these models can capture semantic similarities of code, and maintain high semantic

consistency even in the presence of renaming, statement rearrangement, function migration, or API replacement. GraphSPD [24] leverages graph structures and rich code semantics to detect security patches within code repos. However, it only identifies patch patterns locally and lacks an ecosystem model unifying patch semantics, component dependencies, version evolution, and cross-component vulnerability propagation. Our work extends patch semantic signals to a full supply-chain tracking framework beyond GraphSPD's local code analysis.

In addition, many studies on patch semantic pattern abstraction, patch description generation, and vulnerability feature mining have emerged in the field of patch analysis. Especially for the identification of "Silent Fixes", recent works [24] have significantly improved the accuracy of detecting hidden security patches using graph neural networks. However, these studies still have limitations. Most works focus on a single code repository, a single project, or a single vulnerability, lacking a unified semantic representation mechanism across components and versions, and fail to solve the most challenging task in supply chain scenarios—how to identify semantically equivalent patches among a large number of forks, branches, and downstream modified versions [25]. Therefore, although code semantic modeling lays the foundation for supply chain vulnerability analysis, it is still insufficient to solve the global problem of supply chain-level vulnerability propagation analysis.

2.3. Version Evolution Analysis and Vulnerability Propagation

Research on version evolution analysis and vulnerability propagation has received widespread attention in recent years. Relevant works [26] mainly focus on the introduction, propagation, repair, and regression of vulnerabilities in different versions, and infer the vulnerability life cycle through commit diff, historical record analysis, and rule-based pattern matching. Some studies focus on analyzing the propagation patterns of major vulnerabilities in ecosystems to show the diffusion speed and scope of vulnerabilities among different projects.

Recent studies have begun to attempt to construct "Evolutionary Dependency Graphs" to characterize the dynamic changes of software ecosystems. For example, studies on the npm and Debian ecosystems [27,28] have revealed how vulnerabilities lurk and spread among different versions through transitive dependencies. At the same time, cross-branch and cross-fork version analysis studies [29] have revealed how open-source projects generate multiple variants during long-term evolution, as well as the inheritance and divergence between different variants.

However, most existing studies rely on text-level comparisons or rule-based logic, making it difficult to handle complex evolutionary scenarios in the real world, such as structural transformations, API migration, or semantically equivalent deformation. In addition, most works are case studies, lacking an analysis framework that can be automatically applied to large-scale supply chain environments. Existing methods also do not uniformly model version evolution relationships with code semantic information, patch logic, or dependency chain structures, and thus cannot support holistic analysis across projects, components, and versions. Yet existing constructions only utilize metadata, commit logs, and rule-based methods, ignoring code patch semantics and semantic equivalence between upstream patches and backported fixes. Our CDEG advances this paradigm by introducing semantic nodes and propagation edges derived from hybrid patch representations, connecting vulnerability repair logic to dependency and version evolution paths.

3. Method

This chapter details the proposed software supply chain vulnerability tracking framework based on *LLM-Enhanced Component Dependency Evolution Graph (LLM-Enhanced CDEG)*.

The framework aims to address the failure of traditional SCA tools in the face of code refactoring, version backporting, and silent fixes. Figure 2 presents the overall architecture of the proposed LLM-Enhanced CDEG framework, which comprises three modules: the Code Semantic Understanding Module, the Semantic Fuzzy Matching Module, and the Evolution Graph-Based Impact Analysis Module.

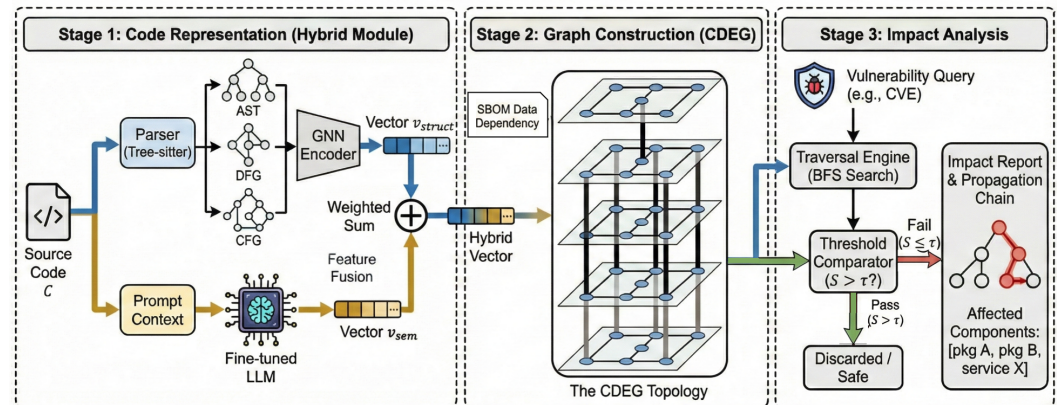


Figure 2. Overall architecture of the LLM-Enhanced CDEG framework. The framework consists of three core modules: Code Semantic Understanding Module, Semantic Fuzzy Matching Module, and Evolution Graph-based Impact Analysis Module. The dashed box indicates the data flow between modules.

3.1. Code Semantic Understanding and Representation

In complex supply chain scenarios, vulnerable code often undergoes transformations such as variable renaming, control flow fine-tuning, or function inlining. To capture the essential logic of code rather than surface-level text, this study constructs a *Structure-Semantic Hybrid Representation* model.

3.1.1. Structural Feature Extraction

We first use Tree-sitter to parse the source code C into an Abstract Syntax Tree (AST), and extract Control Flow Graph (CFG) and Data Flow Graph (DFG). Drawing on the design philosophy of GraphCodeBERT, we focus on data dependency relationships between variables, as these relationships usually remain unchanged during code refactoring.

We map code snippets to a structural feature space $\mathcal{X}_{struct}$. For a given function f , its structural embedding vector $\mathbf{v}_{struct}$ is defined as:

$$\mathbf{v}_{struct} = \text{GNN}_{\phi}(G_{DFG}) \oplus \text{Seq}(\text{AST}_{paths}) \quad (1)$$

where GNN_{ϕ} is a graph neural network-based encoder for aggregating node information in the data flow graph G_{DFG} ; Seq denotes the serial encoding of key paths in the AST; and $\oplus$ represents the vector concatenation operation.

3.1.2. LLM-Based Semantic Embedding

Although structural features are robust, they are difficult to capture the “intent” and high-level logic of code. To this end, we introduce a fine-tuned Large Language Model (LLM) as the semantic encoder. We input the context and diff of the code to obtain its deep semantic vector $\mathbf{v}_{sem}$:

$$\mathbf{v}_{sem} = \text{Transformer}_{\theta}(C, \text{Prompt}_{ctx}) \quad (2)$$

The final code representation vector $\mathbf{F}$ is a weighted fusion of structural and semantic features, defined as:

$$\mathbf{F}(C) = \mathbf{W}_s \cdot \mathbf{v}_{struct} + \mathbf{W}_l \cdot \mathbf{v}_{sem} + \mathbf{b} \quad (3)$$

where $\mathbf{W}_s$ and $\mathbf{W}_l$ are learnable weight matrices used to dynamically adjust the importance of structural and semantic information during training.

3.1.3. Patch Semantics Modeling

To accurately track vulnerability fixes, we need to model patches. We define the semantic change caused by a patch as the difference between the pre-patch and post-patch code vectors, i.e., the *Patch Delta Vector* $\Delta\mathbf{P}$:

$$\Delta\mathbf{P} = \mathbf{F}(C_{post}) - \mathbf{F}(C_{pre}) \quad (4)$$

This vector $\Delta\mathbf{P}$ does not represent specific text line changes, but abstractly characterizes the “fix logic” (e.g., adding a boundary check condition). This representation enables the model to identify the same fix logic implemented in different code forms in downstream versions.

3.2. Semantic Fuzzy Matching and Evolutionary Similarity

Based on the aforementioned hybrid representation, we propose a *Semantic Fuzzy Matching* algorithm to determine whether the target code T contains the semantics of the patch P or retains vulnerability features.

3.2.1. Weighted Similarity Measurement

Traditional tools (e.g., SimHash or Diff) only perform text comparison and are highly susceptible to formatting interference. We define the semantic similarity score $S(P, T)$ between patch P and target function T as follows:

$$S(P, T) = \alpha \cdot \cos(\mathbf{v}_{sem}^P, \mathbf{v}_{sem}^T) + \beta \cdot \text{Sim}_{struct}(\mathbf{v}_{struct}^P, \mathbf{v}_{struct}^T) + \gamma \cdot \mathbb{I}_{API}(P, T) \quad (5)$$

where: $\cos(\cdot, \cdot)$ denotes cosine similarity; Sim_{struct} denotes the structural similarity approximately calculated based on Graph Edit Distance; $\mathbb{I}_{API}$ is an indicator function, which is 1 if both call the same critical sensitive API, otherwise 0; α, β, γ are hyperparameters satisfying $\alpha + \beta + \gamma = 1$. In our experiments, we set $\alpha = \beta = \gamma = 1/3$ as an initial baseline configuration, assigning equal importance to semantic embeddings, structural similarity, and API matching. This setting avoids introducing vulnerability-type-specific tuning at the current stage. We acknowledge that different vulnerability categories may depend on different evidence sources, and, therefore, the equal-weight setting may not be optimal for all types of vulnerabilities.

When $S(P, T) > \tau$ (τ is the decision threshold), we determine that the target code T has the semantics of the patch P , i.e., the vulnerability has been fixed. This fuzzy matching mechanism is particularly effective for identifying Backported Patches in Linux distributions, as these patches usually modify a large number of syntactic structures to adapt to older kernels, but the core fix logic (vector direction) remains consistent.

3.2.2. Evolutionary Consistency Analysis

In cross-version tracking, we further define *Evolutionary Similarity* to detect state jumps of vulnerabilities during version iterations. For consecutive versions v_t and v_{t+1} , their evolutionary differences should be smooth. If the following is detected:

$$\|\mathbf{F}(v_{t+1}) - \mathbf{F}(v_t)\| \gg \epsilon \quad (6)$$

$$\cos(\mathbf{F}(v_{t+1}) - \mathbf{F}(v_t), \Delta\mathbf{P}) \approx 1 \quad (7)$$

it indicates that version v_{t+1} has introduced changes similar to the patch $\Delta\mathbf{P}$, i.e., the system automatically identifies a potential “silent fix” or “patch merge” event.

3.3. Evolution Graph–Based Vulnerability Impact Analysis

To extend micro-level code analysis to macro-level supply chain governance, we construct the *Component Dependency Evolution Graph* (CDEG). CDEG is an extension of traditional static dependency graphs, introducing a time dimension (version evolution) and semantic dimension (vulnerability propagation). The need for graph-level propagation analysis is further illustrated in Figure 3, which compares traditional metadata-based SCA with semantic CDEG analysis in a backporting scenario.

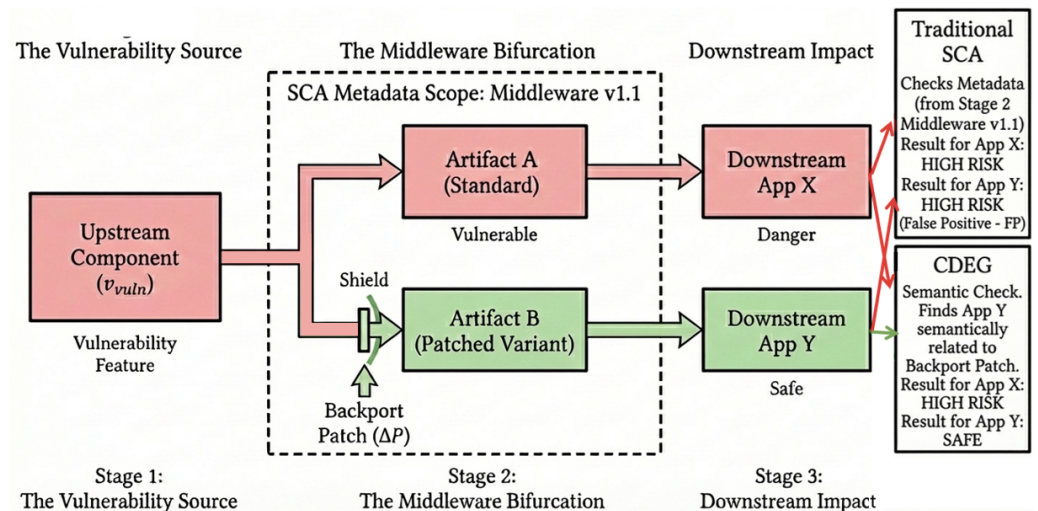


Figure 3. Comparative Illustration of Vulnerability Propagation Analysis: Traditional Metadata-Based SCA vs. Semantic-Based CDEG in Backporting Scenarios.

3.3.1. Graph Formalization

We formalize CDEG as a multi-attribute heterogeneous graph $G = (V, E)$, where: Node set $V = V_C \cup V_V \cup V_S$:

- V_C : Component nodes, representing abstract software packages (e.g., log4j).
- V_V : Version nodes, representing specific release versions (e.g., log4j – 2.14.1).
- V_S : Semantic nodes, storing the hybrid code representation vector $\mathbf{F}$ and patch feature $\Delta\mathbf{P}$ corresponding to the version.

Edge set $E = E_{dep} \cup E_{evol} \cup E_{prop}$:

- E_{dep} : Static dependency edges (Depends-on), derived from SBOM parsing.
- E_{evol} : Version evolution edges (Evolves-to), connecting consecutive versions of the same component.
- E_{prop} : Semantic propagation edges (Propagates-to), dynamically generated by the semantic matching algorithm, representing the implicit flow of vulnerabilities or patches between different components.

For a source patch semantic representation P and a candidate target version v , the system first selects candidate targets using observable supply-chain evidence, including dependency adjacency, version lineage, code reuse or copied components, shared security-sensitive API usage, and package namespace similarity. For each candidate target, the semantic matching score $S(P, T_v)$ is computed using the hybrid similarity function defined in Section 3.2.1. An edge $e = (P, v)$ is added to E_{prop} only when $S(P, T_v) \geq \tau$. In our experiments, τ is set to 0.75, consistent with the validation-optimized global decision threshold. The edge weight $w(e)$ is defined as $S(P, T_v)$, representing the confidence of semantic propagation.

3.3.2. Impact Domain Propagation Algorithm

Given a newly discovered vulnerability patch feature $\Delta \mathbf{P}_{vuln}$, impact domain analysis is transformed into a reachability search problem on the graph. We define the *Impact Set* $\mathcal{I}$ as:

$$\mathcal{I} = \{v \in V_V \mid \exists p \in V_{patch}, \text{Reach}(p, v) \wedge S(p, v) < \tau_{fix}\} \quad (8)$$

where $\text{Reach}(p, v)$ indicates that there is a dependency path from the vulnerability source p to the target version v , and $S(p, v) < \tau_{fix}$ indicates that the target version does not yet contain the semantics of the fix patch. The algorithm is shown as follow (Algorithm 1):

Algorithm 1 Generation of Semantic Propagation Edges E_{prop}

Require: Patch semantic representation P , CDEG $G = (V, E)$, candidate version set V_v , threshold τ_{prop}

Ensure: Semantic propagation edge set E_{prop}

```

1:  $E_{prop} \leftarrow \emptyset$ 
2: for each target version  $v \in V_v$  do
3:   if  $v$  is not connected to  $P$  by dependency, version-lineage, code-reuse, API, or
     namespace evidence then
4:     continue
5:   end if
6:   Extract target code representation  $T_v$ 
7:   Compute semantic matching score  $s \leftarrow S(P, T_v)$ 
8:   if  $s \geq \tau_{prop}$  then
9:     Add edge  $e = (P, v)$  to  $E_{prop}$  with weight  $w(e) \leftarrow s$ 
10:  end if
11: end for
12: return  $E_{prop}$ 

```

By executing Breadth-First Search (BFS) on CDEG combined with semantic filtering, the system can automatically output a complete *Vulnerability Propagation Chain*:

$$\text{Chain} : \text{UpstreamLib}_{v1} \xrightarrow{dep} \text{Middleware}_{v2} \xrightarrow{copy} \text{ProductImage}_{v3} \quad (9)$$

This not only solves the pain point that SBOM cannot identify transitive risks but also eliminates false positives such as “components are introduced but vulnerable functions are not called” through semantic verification. During propagation analysis, error accumulation is controlled by three mechanisms. First, candidate filtering is performed before generating E_{prop} , so semantic propagation edges are created only when observable supply-chain evidence exists. Second, each E_{prop} edge is associated with a confidence score $w(e) = S(P, T_v)$, allowing low-confidence semantic links to be filtered. Third, when a target version is visited during impact analysis, the system re-checks the local fix semantics using the threshold τ_{fix} before treating the version as affected. In the current experiments, τ_{fix} is set to the same validation-optimized threshold as τ_{prop} .

Complexity and Scalability Analysis. Let $n = |V|$ and $m = |E|$ denote the number of nodes and edges in CDEG, respectively. For a specific vulnerability propagation query, BFS-based impact analysis only traverses the reachable subgraph related to the vulnerability source. If this reachable subgraph contains n_r nodes and m_r edges, the time complexity of one propagation query is $O(n_r + m_r)$, and the memory overhead for the queue and visited set is $O(n_r)$. The topology storage cost of CDEG is $O(n + m)$. In addition, if n_s semantic/code representation nodes are stored and the embedding dimension is d , the semantic representation storage cost is $O(n_s d)$. The construction of E_{prop} does not perform exhaustive all-pairs semantic comparison over the entire ecosystem. Instead, candidate targets are first restricted using observable evidence, including dependency adjacency,

version lineage, package metadata, code-reuse evidence, and shared API usage. The hybrid semantic matching score is then computed only for the candidate set. This avoids the prohibitive $O(n^2d)$ cost of all-pairs semantic matching and allows E_{prop} to be updated incrementally when new package versions or patches are introduced.

4. Experiments

4.1. Research Questions

To systematically evaluate the proposed supply chain vulnerability analysis framework based on hybrid semantic representation and Component Dependency Evolution Graph (CDEG), we design the following four Research Questions (RQ):

- **RQ1 (Semantic Patch Detection):** When facing complex scenarios such as code refactoring and backporting, does the hybrid semantic representation of our framework have higher precision and robustness than static methods like VUDDY [30]?
- **RQ2 (Evolutionary Tracking):** In cross-version evolution analysis, can our framework stably track the introduction and fix of vulnerabilities across different commits?
- **RQ3 (Container Image & Instance Impact Analysis):** In real container image supervision scenarios, combined with CDEG, can our framework accurately identify the real state of vulnerabilities in downstream images and deployment instances, solving the false positive and false negative problems of traditional SCA tools caused by “version backporting” or “distribution customization”?
- **RQ4 (Ablation Study):** What are the respective contributions of structural features (Structure), semantic features (LLM), and graph structure (Graph) in the hybrid representation to the system performance?

4.2. Dataset and Setup

4.2.1. Data Collection Sources

To construct a high-credibility experimental benchmark, our data collection covers international general vulnerability databases and Chinese national-level vulnerability databases and focuses on security announcements of mainstream Linux distributions, ensuring the diversity and authority of data sources:

- **Vulnerability Metadata:** We integrate vulnerability descriptions and metadata from NVD (National Vulnerability Database), CVE (Common Vulnerabilities and Exposures), CNVD (China National Vulnerability Database), and CNNVD (China National Vulnerability Database of Information Security).
- **Patch Sources:** Patch codes are collected not only from upstream official repositories but also from security announcements of major Linux distributions and open-source communities, including Ubuntu, OpenEuler, and RedHat.

The SC-Vuln-Real dataset was constructed from both public vulnerability sources and enterprise-assisted verification records. Public sources include vulnerability metadata from NVD, CVE, CNVD, and CNNVD, as well as patch information and security advisories from upstream repositories and major Linux distributions, including Ubuntu, OpenEuler, and RedHat. In addition, enterprise collaboration was used to verify package-version mappings, downstream backport relationships, and container-image vulnerability states in realistic deployment environments. The enterprise-assisted part contains private repository records, internal package metadata, and expert audit labels, and therefore cannot be publicly released.

4.2.2. Dataset Construction: SC-Vuln-Real

Based on the above data sources, we construct the SC-Vuln-Real dataset, which is divided into two parts: “Patch Semantic Dataset” and “Real Container Image Dataset”,

corresponding to the “semantic source” and “analysis object” of graph construction, respectively. The construction of SC-Vuln-Real follows a multi-stage workflow. First, we collect vulnerability records from NVD, CVE, CNVD, and CNNVD, and extract the corresponding CVE identifiers, affected packages, vulnerable version ranges, and advisory descriptions. Second, we retrieve upstream security patches and downstream distribution patches from Ubuntu, OpenEuler, RedHat, and official project repositories. Third, each patch is normalized by removing formatting-only changes, aligning file paths and function names, and extracting pre-patch and post-patch code fragments. Fourth, we identify semantic variants such as backported patches, renamed variables, adjusted control-flow structures, and distribution-specific adaptations. Fifth, duplicate patch records are removed using CVE identifiers, package names, upstream commit IDs, normalized patch hashes, and affected function signatures. Finally, security experts verify the patch labels and container-image vulnerability states by combining official advisories, package metadata, and code-level inspection.

1. *Patch Semantic Dataset (PSD)*: To verify the ability to identify complex patch variants (Semantic Variants) in RQ1 and RQ2, we build a large-scale patch use case library with a total of 18,156 real patch samples. These samples cover diverse backport strategies from upstream communities to downstream distributions, with the following specific distribution:

- Ubuntu Ecosystem: 12,411 patches;
- OpenEuler Ecosystem: 4483 patches;
- RedHat Ecosystem: 1262 patches.

Each PSD record contains the following fields: CVE identifier, package name, distribution source, upstream patch reference, downstream patch reference when available, affected version range, file path, function name, normalized patch hash, patch type label, and semantic-variant label. Each CIID record contains the image identifier, base distribution, package list, dependency information, candidate vulnerability identifier, expert-verified vulnerability state, and evidence source used for verification. These fields are used internally to construct the hybrid patch representation and to evaluate semantic patch detection and vulnerability impact analysis. These patch samples include a large number of code refactoring, variable renaming, and logic adaptation for older kernels/libraries, making them an ideal benchmark for testing semantic fuzzy matching capabilities.

Data Partitioning and Leakage Control. To avoid data leakage in semantic patch detection, we split the Patch Semantic Dataset (PSD) using a group-based strategy rather than a random sample-level strategy. Patch records are grouped by CVE identifier, upstream patch or commit ID, package name, and normalized patch hash. All variants derived from the same upstream fix, including downstream backports, formatting changes, renamed variables, and distribution-specific adaptations, are assigned to the same split. This prevents different versions of the same semantic patch from appearing simultaneously in the training and test sets. The validation set is used for model selection, threshold optimization, and early stopping, while the test set is used only for final evaluation.

2. *Container Image Impact Dataset (CIID)*: To verify the ability to locate vulnerability impact domains in real production environments in RQ3, we build a test dataset containing 297 real Container Images:
 - *Dataset Composition*: The 297 Container Images are collected from real private repositories and Docker Hub, covering base images and business application images with complex dependency chains.

- *Ground Truth*: We organize security experts to conduct detailed manual audits on these 297 Container Images, combining distribution official announcements and code-level verification to establish the real state of vulnerabilities in each image (Ground Truth) for evaluating the final accuracy of each algorithm.

4.2.3. Baselines Selection

To comprehensively verify the effectiveness of the proposed method, we compare with the following five representative baseline technologies:

1. *OWASP Dependency-Check (OWASP DC)*: Parses version numbers based on SBOM and matches NVD. Represents the mainstream component analysis method in current enterprises [14].
2. *SimHash*: Locality-sensitive hashing algorithm based on text features. Represents traditional fuzzy matching technology [7].
3. *VUDDY*: Locality-sensitive hashing technology at the function level. Represents the highest level of exact matching in current code clone detection [30].
4. *GraphCodeBERT*: Pre-trained code model released by Microsoft. Represents the effect of pure semantic vectors without introducing the “structural-semantic hybrid representation” of this study [22].
5. *LLM Zero-shot*: Directly uses general-purpose LLM for judgment. Used to evaluate the general intelligence level without specialized architecture.[31].

4.2.4. Implementation Details

For the Code Semantic Understanding Module, we adopted Qwen3-32B [32] as the base semantic encoder. The Patch Semantic Dataset was split into training, validation, and test sets at a ratio of 8:1:1 using the group-based partitioning strategy described above. The base model was loaded in 4-bit NormalFloat (NF4) precision, and QLoRA adapters were injected into the attention modules and feed-forward networks. The model was trained for 8 epochs with an initial learning rate of 1×10^{-5} and a batch size of 16. The LoRA rank was set to $r = 64$, and the LoRA scaling parameter was set to $\alpha = 16$. We used Adam as the optimizer. The stopping criterion was based on the best validation F1 score, with early stopping monitored on validation loss. The global decision threshold was optimized on the validation set and set to $\tau = 0.75$. All experiments were conducted on an Ubuntu 20.04 server equipped with 4 NVIDIA RTX 4090 GPUs. QLoRA was selected because full fine-tuning of a 32B-parameter model is impractical under our available GPU-memory budget. By loading the base model in 4-bit NF4 precision and training only low-rank adapter parameters, QLoRA substantially reduces memory consumption while retaining the ability to adapt the model to the patch semantic analysis task. Compared with prompt-only or zero-shot inference, QLoRA enables the semantic encoder to learn task-specific patch-delta representations from PSD. Therefore, QLoRA provides a practical balance between adaptation capability, training cost, and hardware feasibility.

4.3. RQ1: Robustness of Semantic Patch Detection

Goal: Verify the patch detection capability at the micro level, especially when facing code transformations (Refactoring/Backport).

Comparison Strategy: Focus on comparing VUDDY (weak anti-deformation ability), GraphCodeBERT (pure semantics), and LLM Zero-shot (hallucination problem existing). We test the performance of each method on the PSD dataset, which contains approximately 40% Semantic Variants (i.e., patches that have undergone refactoring, renaming, or Backport). Besides the zero-shot LLM baseline, we adopt an ablation variant “w/o Structure (Only LLM)” in Section 4.6 to quantify QLoRA adaptation gain. This variant

uses QLoRA-tuned Qwen3-32B but discards AST/CFG/DFG structural branches, relying solely on semantic embeddings for patch matching. Comparing zero-shot LLM with this variant reveals the gain from task-specific QLoRA tuning, whereas contrasting it with our full model demonstrates the extra performance brought by hybrid structure-semantic representations.

Table 1 reports the patch detection results on the PSD benchmark and highlights the robustness of different methods under semantic variants such as refactoring and backporting. *Result Analysis:*

1. *Limitations of VUDDY:* VUDDY performs excellently when the code is unchanged (Direct Merge) (Recall = 0.98), but its Recall drops sharply to 19% in Semantic Variant scenarios. This proves that pure hashing methods cannot cope with common code customization in the supply chain.
2. *Precision Trade-off of LLM Zero-shot:* Although GPT-4 achieves a high recall rate (79.0%) in variant identification, its Precision is low (0.76), tending to misjudge code that only contains similar variable names but completely different logic as fixed (False Positive).
3. *Advantages of Our Method:* By combining structural features (AST/CFG) and semantic embedding, our method not only maintains high precision close to VUDDY (0.92) but also achieves a recall rate of 0.88 in variant identification, significantly outperforming GraphCodeBERT and Zero-shot methods. This indicates that “structural constraints + semantic understanding” is the key to solving fuzzy patch matching.

Table 1. Patch Detection Performance with 95% Confidence Intervals.

Method	Overall F1	Precision	Recall (DM)	Recall (SV)
SimHash	0.47	0.92 ± 0.004	0.45 ± 0.009	0.12 ± 0.007
VUDDY	0.79	0.99 ± 0.002	0.98 ± 0.003	0.19 ± 0.009
GraphCodeBERT	0.80	0.81 ± 0.006	0.88 ± 0.006	0.65 ± 0.011
LLM Zero-shot	0.81	0.76 ± 0.006	0.92 ± 0.005	0.79 ± 0.009
Ours (Hybrid Repr.)	0.92	0.92 ± 0.004	0.94 ± 0.004	0.88 ± 0.007

4.4. RQ2: Stability of Evolutionary Tracking

Goal: Verify the ability to track vulnerabilities over a long time span.

Comparison Strategy: Compare SimHash (representing the traditional text methods) and GraphCodeBERT (representing the semantic methods).

We define the *Evolution Consistency (EC)* metric to quantify the ability of the detector to maintain stable judgments in a sequence of consecutive versions. For a component evolution chain $V = \{v_1, v_2, \dots, v_T\}$ containing T consecutive versions, let $D(v_i) \in \{0, 1\}$ denote the vulnerability detection result of the model for version v_i . EC is defined as follows:

$$EC = 1 - \frac{\sum_{t=1}^{T-1} \mathbb{I}(D(v_t) \neq D(v_{t+1}))}{T - 1} \quad (10)$$

where $\mathbb{I}(\cdot)$ is the indicator function. This metric penalizes unexpected state flips between adjacent versions.

Data Summary: As illustrated in Figure 4, we visualize the distribution of Evolution Consistency (EC) scores using violin plots to demonstrate the stability variance across different methods. The dashed line indicates the system’s confidence threshold ($\tau = 0.75$).

- *SimHash (mean EC = 0.62):* The violin plot for SimHash exhibits a wide and deeply stretched distribution, with a massive portion of its scores falling well below the 0.75 confidence threshold. This visually confirms the severe “flicker” phenomenon—minor code formatting or comment modifications cause the hashing similarity to randomly drop below the threshold, leading to heavily broken propagation chains.

- *GraphCodeBERT* (mean EC = 0.84): While its distribution is more concentrated than SimHash, GraphCodeBERT still presents a noticeable lower tail extending below the threshold. In cases of large-scale refactoring (e.g., function logic moved across files), pure semantic vectors struggle to maintain high consistency, occasionally resulting in tracking loss
- *Ours* (EC = 0.96): Benefiting from the Evolution Edge in the CDEG and the Semantic Delta Vector (ΔP), our method demonstrates extreme stability. The distribution is highly concentrated near 1.0, safely above the 0.75 threshold. Even when code undergoes structural migration, the graph topology securely locks the vulnerability logic. The extremely rare outliers (represented by the isolated dots) correspond to highly irregular, massive architectural rewrites, yet the overall system remains highly robust.

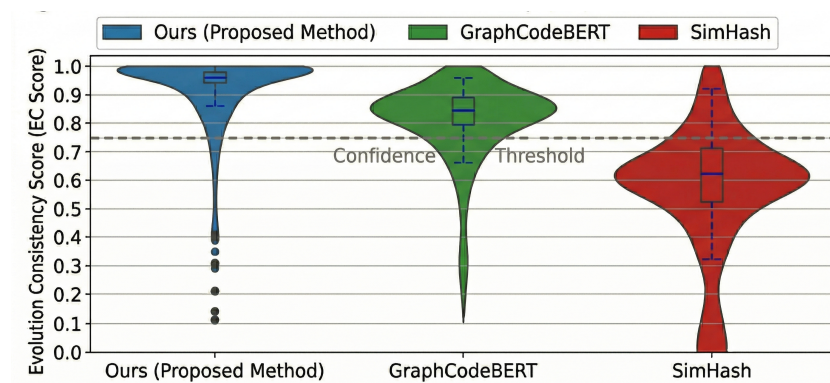


Figure 4. Comparison of Evolution Consistency (EC) metric across different methods.

4.5. RQ3: Container Image & Instance Impact Analysis

Goal: This experiment is key to verifying the practical application capability of the framework. We evaluate whether the system can correctly identify the security state of components in images by combining the knowledge of the PSD patch library in the CIID dataset (297 real Container Images).

Comparison Strategy: Focus on comparing OWASP DC (version matching, industry benchmark) and VUDDY (exact hashing, static detection benchmark).

Table 2 summarizes the container-image impact analysis results on 297 images and compares version matching, exact hashing, pretrained semantic matching, and the proposed CDEG-based method. *In-depth Analysis:*

1. **Accuracy Breakthrough (High Accuracy):** Among the 297 tested Container Images, our system achieves accurate judgment for 279, with an F1 score of 93.9%. In contrast, the traditional OWASP DC can only correctly get 67%.
2. **FPR Reduction:** The FPR of OWASP DC is as high as 54.4%, mainly because mainstream distributions (e.g., Ubuntu, OpenEuler) generally adopt the “version freezing, patch backporting” strategy (e.g., openssl-1.1.1f contains the patch for CVE-2020-1967, but the version number remains unchanged). Our system injects the semantics of 18,000 distribution-specific patches through the PSD dataset, and CDEG successfully matches these “silent fixes”, reducing the FPR to 4.8%.
3. **FNR Reduction:** The FNR of VUDDY is as high as 54.7%, mainly because container image construction often involves compilation parameter adjustment, stripped binaries, or file system hierarchy flattening, which destroys the premise of exact hashing. Through structured semantic embedding and context inference of the dependency graph, our system can still locate vulnerabilities along the dependency chain even when the binary fingerprint changes, controlling the FNR to 7.3%.

Table 2. Impact Analysis Performance on 297 Container Images with 95% Confidence Intervals. Up/down arrows means the larger/lower value is better.

Method	F1-Score (↑)	Precision (↑)	Recall (↑)	FPR (↓)	FNR (↓)
OWASP DC	67.0%	59.2% ± 5.5%	77.3% ± 4.9%	54.4% ± 5.7%	22.7% ± 4.9%
VUDDY	58.9%	84.0% ± 4.3%	45.3% ± 5.7%	8.8% ± 3.4%	54.7% ± 5.7%
GraphCodeBERT	85.6%	84.0% ± 4.3%	87.3% ± 3.9%	17.0% ± 4.4%	12.7% ± 3.9%
Ours (CDEG)	93.9%	95.2% ± 2.6%	92.7% ± 3.1%	4.8% ± 2.6%	7.3% ± 3.1%

4.6. RQ4: Ablation Study (Component Effectiveness Analysis)

Goal: Verify the independent contributions of the two core components—“Hybrid Representation (HR)” and “CDEG Propagation (Graph)”—to the system performance.

Variant Definitions:

1. *w/o Structure (Only LLM):* Remove AST/CFG structural features, only use semantic vectors generated by Qwen3.
2. *w/o LLM (Only Structure):* Remove the LLM component, only use structured embedding based on GraphCodeBERT (similar to enhanced VUDDY).
3. *w/o Graph (No Prop.):* Remove the CDEG graph propagation module, only perform isolated semantic comparison for each component without considering dependency chain context.

Table 3 presents the ablation results used to isolate the contributions of structural features, LLM-based semantic features, and CDEG graph propagation. **Result Analysis** Based on the ablation experiment results, we analyze the contribution of each core component. It is noted that the F1-Score here is calculated strictly based on the Semantic Variant (SV) Recall to highlight robustness against code changes.

1. *w/o Structure (Only LLM):* Compared with the full system, its Precision drops significantly from 0.92 to 0.78, while Recall remains relatively high (0.85). This is because without the syntactic constraints of AST/CFG structural features, the LLM is prone to “hallucination”—misjudging code that only modifies variable names but does not fix the core logic as having been repaired, leading to more False Positives.
2. *w/o LLM (Only Structure):* Its Recall plummets from 0.88 to 0.61, and F1-Score drops to 0.72. Structural features are highly sensitive to large-scale refactoring during Backport (e.g., function inlining, code cross-file migration), making it impossible to recognize semantically equivalent but structurally distinct patch variants, resulting in a large number of False Negatives.
3. *w/o Graph (No Prop.):* Its single-point detection performance (Precision = 0.91, Recall = 0.88) is close to the full system, but it faces “systemic failure” in supply chain scenarios. Without the CDEG graph to prune “unreachable components”, the FPR surges from 4.8% to 28.3% in the CIID dataset. This proves that the graph propagation module is crucial for reducing false positives in large-scale supply chain analysis, even if it contributes little to single-component detection.

Table 3. Ablation Study: Contribution of Core Components. Up/down arrows means the larger/lower value is better.

Model Variant	F1-Score (↑)	Precision (↑)	Recall (SV) (↑)	CIID FPR (↓)
Full System (Ours)	0.90	0.92 ± 0.004	0.88 ± 0.007	4.8% ± 2.6%
w/o Structure (Only LLM)	0.81	0.78 ± 0.006	0.85 ± 0.008	6.2% ± 2.9%
w/o LLM (Only Struct.)	0.72	0.89 ± 0.005	0.61 ± 0.011	15.4% ± 4.2%
w/o Graph (No Prop.)	0.89	0.91 ± 0.004	0.88 ± 0.007	28.3% ± 5.2%

Conclusions:

- *Structural Features (Structure)* are the key to ensuring high Precision. They provide necessary syntactic constraints for LLM reasoning and suppress hallucinations.
- *LLM Semantic Features* are the key to ensuring high Recall. They enable the system to understand Backport code with “semantic equivalence but distinct structures”.
- *CDEG Graph* contributes little to single-point detection but is crucial for reducing the FPR of the entire supply chain analysis.

4.7. Sensitivity Analysis

To evaluate the robustness of our framework and justify the choice of the optimal decision threshold τ , we conduct a sensitivity analysis on the Patch Semantic Dataset (PSD). The threshold τ (ranging from 0.0 to 1.0) dictates whether a target code snippet T contains the patched semantic logic P . We vary τ with a step size of 0.05 and measure the variations in Precision, Recall on Semantic Variants (SV), and the Overall F1-Score.

As shown in Figure 5, when τ increases, the system applies stricter constraints to match the semantic and structural vectors.

- **Low Threshold Domain ($\tau < 0.50$):** When the threshold is set too low, the system achieves a near-perfect Recall ($\approx 98\%$) but suffers from a severely degraded Precision ($\approx 64\%$). This occurs because the fuzzy matching mechanism incorrectly flags refactored codes with similar contextual text or API definitions as fixed, leading to a high False Positive Rate (FPR).
- **High Threshold Domain ($\tau > 0.85$):** Conversely, as τ approaches 1.0, the matching criteria shift closer to exact cloning. While Precision plateaus near 97%, the Recall for Semantic Variants (SV) plummets sharply to below 40%. Under such restrictive bounds, modified code structures from downstream backporting variants fail to exceed the matching line, leading to substantial False Negatives (FN).
- **The Equilibrium Point ($\tau = 0.75$):** The Overall F1-Score forms a parabolic curve that peaks at $\tau = 0.75$ which is marked by red dashed line in Figure 5. At this specific juncture, the system reaches an optimal equilibrium, retaining an impressive Precision of 92% while maintaining a high SV Recall of 88%. This empirically proves that a threshold of 0.75 provides sufficient margin to accommodate semantic deformations during software evolution without introducing catastrophic noise.

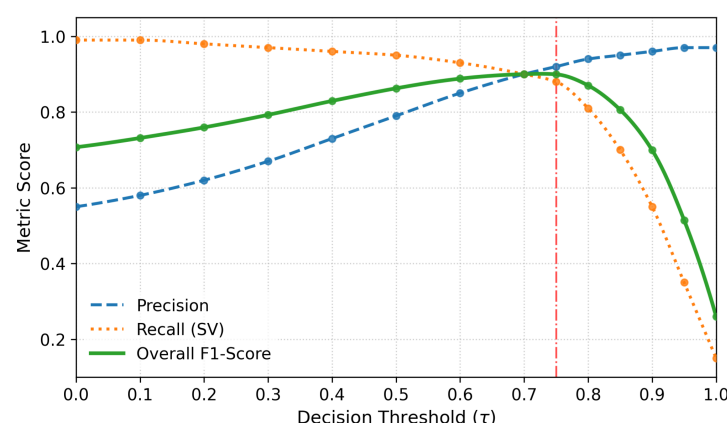


Figure 5. Impact of decision threshold τ on verification performance metrics.

4.8. Discussion

The experimental results clearly define the position of this study in the technical spectrum:

- Compared with VUDDY/SimHash, we solve the “**robustness**” problem and can cope with code variations and build customization in the supply chain.

- Compared with OWASP DC, we solve the “**granularity**” problem, evolving from rough version matching to precise code semantic analysis, effectively addressing distribution patch backporting.
- Compared with LLM Zero-shot, we solve the “**reliability and cost**” problems, reducing hallucinations through hybrid representation and achieving efficient global inference using the CDEG graph.

This series of comparisons strongly proves the unique value and practical capability of the “hybrid semantic representation + evolutionary graph” architecture in solving complex vulnerability governance problems in large-scale software supply chains.

5. Conclusions

5.1. Research Summary

This study addresses the key limitations of traditional software supply chain security tools in handling complex scenarios, such as patch backporting, code refactoring, and silent fixes. We propose an LLM-Enhanced Component Dependency Evolution Graph (CDEG) framework, which integrates structural-semantic hybrid representation and graph-based propagation analysis.

The core contributions including: first, the hybrid representation model combines AST/CFG structural features and LLM semantic embedding, effectively capturing both code syntax and high-level logic; second, the CDEG graph integrates time, space, and semantic dimensions, realizing cross-version and cross-component vulnerability propagation tracking; third, extensive experiments on the SC-Vuln-Real dataset (18,156 patches, 297 container images) verify the framework’s superiority—achieving 88.4% recall for backported patches, 93.9% accuracy for container image analysis, and significantly reducing false positives/negatives compared with baselines.

5.2. Limitations and Future Work

Despite the promising results, the framework still has limitations that point to future research directions:

- *Cross-language Adaptability:* **Cross-language Adaptability:** Although the CDEG abstraction is language-agnostic at the graph level, the current empirical evaluation is limited to C/C++ and Python 3.10. Extending the framework to Java and Go is relatively direct because their dependency metadata and static structures are more regular. Rust requires additional handling of macro expansion, feature flags, and Cargo dependency resolution. JavaScript and TypeScript require support for dynamic imports, transpilation, bundling, minification, and lockfile variants. Therefore, the reported empirical results should be interpreted within the evaluated C/C++ and Python settings, while large-scale validation on Java, Go, Rust, and JavaScript ecosystems remains future work.
- *Real-time and Interpretability:* The CDEG graph relies on periodic updates, and the LLM-based semantic process lacks explicit interpretability. We plan to develop real-time update mechanisms and integrate attention mechanisms to generate human-readable propagation paths.
- *Scalability for Large Ecosystems:* Although the proposed candidate filtering and bounded traversal strategy is designed to support scalable analysis, the current submission does not include quantitative resource-cost analysis on ultra-large ecosystems such as npm, Maven, or PyPI due to dataset-access conditions and computational-resource constraints. These ecosystems introduce additional challenges, including high dependency churn, lockfile diversity, transitive dependency expansion, and large storage

requirements. We have included full-ecosystem deployment and resource-cost profiling on npm, Maven, and PyPI in our future research plan.

- *Vulnerability-type-aware Similarity Weighting:* The current implementation adopts a uniform weighting strategy for semantic similarity, structural similarity, and API matching. However, different vulnerability types may require different evidence distributions. For example, API-misuse vulnerabilities may benefit from higher API-matching weights, memory-safety vulnerabilities may require stronger structural evidence, and logic-bypass vulnerabilities may rely more on semantic intent. In future work, we will investigate adaptive or learned weighting strategies based on vulnerability categories such as CWE classes.
- *Stronger Baseline Coverage:* The current evaluation includes GraphCodeBERT as a representative pretrained code-representation baseline, LLM Zero-shot as a representative direct LLM baseline, and ablation variants to evaluate the contribution of the proposed CDEG components. However, it does not exhaust all recent graph-based vulnerability detection methods or retrieval-augmented LLM systems. Future work will construct fair comparison protocols for these newer methods by aligning input formats, retrieval corpora, retrieval budgets, prompt settings, ground-truth labels, and evaluation metrics with the PSD and CIID tasks. We will also explore stronger contemporary models to further improve the overall effectiveness of the system and to validate the applicability of CDEG across different model backbones and analysis settings.

Author Contributions: Conceptualization, Y.G. and H.W.; methodology, Y.G., H.W., and Y.L.; software, Y.G., H.W., and Y.L.; validation, Y.G., P.S., and W.S.; formal analysis, Y.G.; investigation, Y.G.; resources, J.C.; data curation, Y.G.; writing—original draft preparation, Y.G.; writing—review and editing, H.W. and P.S.; visualization, Y.G. and W.S.; supervision, J.C.; project administration, J.C.; funding acquisition, J.C. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding

Data Availability Statement: The data presented in this study are not publicly available due to commercial confidentiality, privacy, and security restrictions. Specifically, part of the dataset was derived from real enterprise deployment and operational environments, including internal container images and vulnerability validation data. Public release of these materials could expose sensitive operational information and therefore is restricted. Researchers may contact the corresponding author to discuss the possibility of access under appropriate conditions and restrictions.

Acknowledgments: AI-assisted tools were used in the preparation of the manuscript for language translation.

Conflicts of Interest: Authors Yuanjun Gao, Hongzhou Wu, Yujia Luo and Peichang Shi were employed by the KylinSoft Corporation. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

SBOM	Software Bill of Materials
SCA	Software Composition Analysis
CVE	Common Vulnerabilities and Exposures
NVD	National Vulnerability Database
CNVD	China National Vulnerability Database
CNNVD	China National Vulnerability Database of Information Security
AST	Abstract Syntax Tree
CFG	Control Flow Graph
LLM	Large Language Model

References

1. Lenarduzzi, V.; Taibi, D.; Tosi, D.; Lavazza, L.; Morasca, S. Open source software evaluation, selection, and adoption: A systematic literature review. In Proceedings of the 2020 46th Euromicro Conference on Software Engineering and Advanced Applications (SEAA), Portoroz, Slovenia, 26–28 August 2020; IEEE: New York, NY, USA, 2020; pp. 437–444.
2. Przymus, P.; Durieux, T. Wolves in the Repository: A Software Engineering Analysis of the XZ Utils Supply Chain Attack. In Proceedings of the 2025 IEEE/ACM 22nd International Conference on Mining Software Repositories (MSR), Ottawa, ON, Canada, 28–29 April 2025.
3. Zahan, N.; Lin, E.; Tamanna, M.; Enck, W.; Williams, L. Software Bills of Materials Are Required. Are We There Yet? *IEEE Secur. Priv.* **2023**, *21*, 82–88.
4. Woo, S.; Hong, H.; Choi, E.; Lee, H. MOVERY: A Precise Approach for Modified Vulnerable Code Clone Discovery from Modified Open-Source Software Components. In Proceedings of the 31st USENIX Security Symposium, Security 2022, Boston, MA, USA, 10–12 August 2022.
5. Nguyen, S.; Vu, T.T.; Vo, H.D. VFFINDER: A Graph-Based Approach for Automated Silent Vulnerability-Fix Identification. In Proceedings of the 2023 15th International Conference on Knowledge and Systems Engineering (KSE), Hanoi, Vietnam, 18–20 October 2023.
6. Zhou, J.; Pacheco, M.; Wan, Z.; Xia, X.; Lo, D.; Wang, Y.; Hassan, A.E. Finding A Needle in a Haystack: Automated Mining of Silent Vulnerability Fixes. In Proceedings of the ASE '21: 36th IEEE/ACM International Conference on Automated Software Engineering, Melbourne, Australia, 15–19 November 2021.
7. Charikar, M.S. Similarity Estimation Techniques from Rounding Algorithms. In Proceedings of the 34th Annual ACM (Association for Computing Machinery) Symposium on Theory of Computing, Montreal, QC, Canada, 19–21 May 2002.
8. Yarra, R. LLMpatronous: Harnessing the Power of LLMs For Vulnerability Detection. *arXiv* **2025**, arXiv:2504.18423.
9. Ponta, S.E.; Plate, H.; Sabetta, A. Beyond Metadata: Code-centric and Usage-based Analysis of Known Vulnerabilities in Open-source Software. In Proceedings of the IEEE International Conference on Software Maintenance and Evolution, Madrid, Spain, 23–29 September 2018.
10. Sun, Z.-Y.; Wu, J.-Z.; Ling, X.; Wei, Y.-L.; Luo, T.-Y.; Wu, Y.J. Research on Key Technologies of SBOM in Software Supply Chain. *Ruan Jian Xue Bao/J. Softw.* **2025**, *36*, 2604–2642.
11. Reichert, B.M.; Obelheiro, R.R. Software supply chain security: A systematic literature review. *Int. J. Comput. Appl.* **2024**, *46*, 853–867.
12. Woo, S.; Park, S.; Kim, S.; Lee, H.; Oh, H. CENTRIS: A Precise and Scalable Approach for Identifying Modified Open-Source Software Reuse. In Proceedings of the 2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE) 2021, Madrid, Spain, 22–30 May 2021; pp. 860–872.
13. Ruan, L.; Xu, Q.; Zhu, S.; Huang, X.; Lin, X. A Survey of Binary Code Similarity Detection Techniques. *Electronics* **2024**, *13*, 1715.
14. Ndebugre, M.; Nabil, M.; Patooghy, A.; Sarrafzadeh, A. A Comprehensive Software Vulnerability Dataset Based on OWASP Top Ten Standard. In Proceedings of the 2025 Silicon Valley Cybersecurity Conference (SVCC), San Francisco, CA, USA, 23–25 June 2025.
15. Imtiaz, N.; Thorn, S.; Williams, L. A comparative study of vulnerability reporting by software composition analysis tools. In Proceedings of the ESEM '21: Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM), Bari, Italy, 11–15 October 2021.
16. Sultan, S.; Ahmad, I.; Dimitriou, T. Container Security: Issues, Challenges, and the Road Ahead. *IEEE Access* **2019**, *7*, 52976–52996.
17. Wei, H.; Li, M. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. In Proceedings of the International Joint Conference on Artificial Intelligence, Melbourne, Australia, 19–25 August 2019.
18. Zhou, Y.; Liu, S.; Siow, J.; Du, X.; Liu, Y. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. *Adv. Neural Inf. Process. Syst.* **2019**, *32*, 10197–10207.

19. Yamaguchi, F.; Golde, N.; Arp, D.; Rieck, K. Modeling and Discovering Vulnerabilities with Code Property Graphs. In Proceedings of the 2014 IEEE Symposium on Security and Privacy, Berkeley, CA, USA, 18–21 May 2014; pp. 590–604.
20. Feng, Z.; Gurr, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In Proceedings of the Conference on Empirical Methods in Natural Language Processing, Virtual, 16–20 November 2020.
21. Kanade, A.; Maniatis, P.; Balakrishnan, G.; Shi, K. Learning and Evaluating Contextual Embedding of Source Code. *Proc. Int. Conf. Mach. Learn.* **2021**, *119*, 5110–5121.
22. Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. Graphcodebert: Pre-training code representations with data flow. In Proceedings of the 9th International Conference on Learning Representations, ICLR 2021, Virtual, 3–7 May 2021.
23. Wang Y.; Wang, W.; Joty, S.; Hoi, S.C.H. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In Proceedings of the Conference on Empirical Methods in Natural Language Processing (EMNLP), Punta Cana, Dominican Republic, 7–1 November 2021.
24. Wang, S.; Wang, X.; Sun, K.; Jajodia, S.; Wang, H.; Li, Q. GraphSPD: Graph-Based Security Patch Detection with Enriched Code Semantics. In Proceedings of the 2023 IEEE Symposium on Security and Privacy (SP), San Francisco, CA, USA, 21–25 May 2023.
25. Gkortzis, A.; Mitropoulos, D.; Spinellis, D. VulinOSS: A Dataset of Security Vulnerabilities in Open-Source Systems. In Proceedings of the IEEE/ACM International Conference on Mining Software Repositories, Gothenburg, Sweden, 28–29 May 2018.
26. Li, F.; Paxson, V. A Large-Scale Empirical Study of Security Patches. In Proceedings of the CCS '17: 2017 ACM SIGSAC Conference on Computer and Communications Security, Dallas, TX, USA, 30 October–3 November 2017.
27. Zimmermann, M.; Staicu, C.-A.; Tenny, C.; Pradel, M. Small world with high risks: A study of security threats in the NPM ecosystem. In Proceedings of the 28th USENIX Security Symposium, Santa Clara, CA, USA, 14–16 August 2019.
28. Decan, A.; Mens, T.; Grosjean, P. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empir. Softw. Eng.* **2019**, *24*, 381–416.
29. Businge, J.; Openja, M.; Nadi, S.; Bainomugisha, E.; Berger, T. Clone-Based Variability Management in the Android Ecosystem. In Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution (ICSME), Madrid, Spain, 23–29 September 2018.
30. Kim, S.; Voo, S.; Lee, H.; Oh, H. VUDDY: A Scalable Approach for Vulnerable Code Clone Discovery. In Proceedings of the 2017 IEEE Symposium on Security and Privacy (SP), San Jose, CA, USA, 22–26 May 2017; pp. 595–614.
31. OpenAI. GPT-4 Technical Report. *arXiv* **2023**, arXiv:2303.08774.
32. Qwen Team. Qwen3 Technical Report. *arXiv* **2025**, arXiv:2505.09388, .

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.