An Accurate and Efficient Vulnerability Propagation Analysis Framework

Bonan Ruan Zhiwei Lin Jiahao Liu Chuqi Zhang Kaihang Ji Zhenkai Liang
National University of Singapore
{bonan.ruan,zhiweil,jiahaoliu,chuqiz,kaihang.ji}@u.nus.edu {liangzk}@nus.edu.sg

***Abstract*—Identifying the impact scope and scale is critical for software supply chain vulnerability assessment. However, existing studies face substantial limitations. First, prior studies either work at coarse package-level granularity—producing many false positives—or fail to accomplish whole-ecosystem vulnerability propagation analysis. Second, although vulnerability assessment indicators like CVSS characterize individual vulnerabilities, no metric exists to specifically quantify the dynamic impact of vulnerability propagation across software supply chains. To address these limitations and enable accurate and comprehensive vulnerability impact assessment, we propose a novel approach: (i) a hierarchical worklist-based algorithm for whole-ecosystem and call-graph-level vulnerability propagation analysis and (ii) the Vulnerability Propagation Scoring System (VPSS), a dynamic metric to quantify the scope and evolution of vulnerability impacts in software supply chains. We implement a prototype of our approach in the Java Maven ecosystem and evaluate it on 100 real-world vulnerabilities. Experimental results demonstrate that our approach enables effective ecosystem-wide vulnerability propagation analysis, and provides a practical, quantitative measure of vulnerability impact through VPSS.**

I. Introduction

The proliferation of software vulnerabilities has introduced significant security risks. However, not all vulnerabilities carry the same impact scope. Specifically, a vulnerability in a client application usually only affects the application itself, while a vulnerability in software supply chains often puts downstream software that depends on the vulnerable upstream libraries directly or transitively at risk as well. For instance, *Heartbleed* [1], a vulnerability in the OpenSSL library, affects countless services that rely on it. Another vulnerability, *Log4Shell* [2], endangers numerous projects depending on the popular Apache Log4j logging framework [3]. Hence, it is critical to *identify the impact scope and scale of the vulnerability in software supply chains* after a vulnerability is disclosed, which is called *vulnerability propagation analysis*.

Researchers have conducted several studies to investigate this problem for popular programming language ecosystems (*e.g.,* Java [4]–[18], JavaScript [19]–[23], and Python [24]) by analyzing components and the corresponding dependencies in software supply chains. While existing works take significant steps toward software supply chain vulnerability analysis, a substantial gap remains in enabling accurate and comprehensive impact assessment. We observe the following fundamental limitations that render current solutions suboptimal:

Limitation 1 (L1): The lack of accurate and complete vulnerability propagation analysis. First, many studies only conduct package-level vulnerability propagation analysis based on dependency declarations. This often leads to false positives, as downstream projects may declare dependencies on vulnerable packages without actually invoking the vulnerable functions (VFs) [11], [14], [18]. Second, although the remaining studies have explored call graph (CG)-level analysis, their methods are often limited in scope and incomplete due to a lack of efficient processing techniques for complex whole-ecosystem dependencies. Specifically, (i) they consider only partial dependency relationships rather than the complex, ecosystem-wide structure; (ii) they analyze only a subset of project versions or a limited number of downstream projects, instead of covering all relevant versions and dependencies; and (iii) they focus solely on direct dependencies, ignoring vulnerability propagation through transitive dependencies. We provide a detailed discussion of these limitations in § II-B1.

Limitation 2 (L2): The lack of metrics for quantifying the impact of vulnerabilities across software supply chains. The widely adopted vulnerability assessment indicators are only used to characterize the impact of a vulnerability itself. For instance, although people can perceive the severity of a vulnerability through its CVSS score [25], it is fundamentally designed to assess and reflect the characteristics of individual vulnerabilities. As such, its application does not extend well to measuring vulnerability impacts across software supply chains, which is explicitly acknowledged in the CVSS v4.0 FAQ [26].

To address these limitations, we propose a novel approach to accurate and whole-ecosystem impact assessment of software supply chain vulnerabilities and a propagation-based indicator for quantifying such impact. Specifically, to address **L1**, we draw inspiration from the data flow analysis and design a worklist-based vulnerability propagation analysis algorithm to efficiently identify affected downstream dependencies given a specific vulnerability, which conducts CG-level analysis that considers complete dependency relations, all potentially affected downstreams, and transitive dependencies in a whole ecosystem. Notably, we integrate hierarchical pruning strategies into the propagation algorithm to reduce the complexity of analysis. To address **L2**, we propose the *Vulnerability Propagation Scoring System* (VPSS), a graph-theoretic dynamic indicator specialized for quantifying vulnerability impact in software supply chains and reflecting the temporal evolution of impact. It is designed to consider both the breadth and depth of vulnerability propagation. Furthermore, VPSS has a similar score range (0–10) and impact levels (low, medium, high, and

TABLE I: Term Unification Across Ecosystems

Ecosystem	**Project (*P*)**	**Project-Version (*PV*)**
Maven (Java)	GroupId:ArtifactId	GroupId:ArtifactId:Version
npm (JavaScript)	package name	package@version
PyPI (Python)	distribution name	name==version

critical) as CVSS, making it easy to understand and use.

We implement a prototype of our approach for the Java Maven ecosystem [27], one of the largest software ecosystems in the world, and evaluate it on 100 real-world vulnerabilities investigated by prior work [14]. The evaluation results indicate that our approach successfully and efficiently completes the propagation analysis for all 100 vulnerabilities. On average, 97.8% of projects and 99.2% of project version releases are pruned during the analysis, with the longest and average propagation path lengths reduced by at least 34.1% and 29.4%, respectively. Importantly, our approach significantly lowers the cost of call graph construction by reducing the number of CGs that need to be built. In addition, by computing and analyzing the VPSS scores for these vulnerabilities, we obtain insightful findings: The VPSS scores generally decline over time after disclosure, driven by patch adoption and ecosystem expansion. Interestingly, some CVEs (*e.g.,* CVE-2016-3086 [28]) show temporary score increases due to delayed dependency updates. Across the entire dataset, VPSS scores remain relatively low and gradually stabilize. This aligns with expected ecosystem dynamics where vulnerability propagation attenuates over time. Overall, the results validate that VPSS captures both temporal and distributional aspects of vulnerability impact, offering a practical metric for supply chain risk assessment.

To the best of our knowledge, this is the first work achieving CG-level and whole-ecosystem vulnerability impact assessment for software supply chains. In summary, this paper makes the following contributions:

- We design a hierarchical worklist-based vulnerability propagation analysis algorithm to accurately and efficiently identify affected downstream dependencies across a whole software ecosystem.
- We propose Vulnerability Propagation Scoring System (VPSS), the first time-aware indicator for quantifying vulnerability impact in software supply chains.
- We implement a prototype of our approach for the Java Maven ecosystem and evaluate it on real-world vulnerabilities. We will release our prototype's code upon the paper's acceptance to support future research.

II. Background and Motivations

A. Background

1) Terminology: Different software ecosystems use diverse naming conventions to refer to software units and their versions, such as *packages*, *modules*, or *distributions*. This inconsistency may lead to inaccurate descriptions and hinder understanding. To provide a unified abstraction across ecosystems, we use the terms *Project* (*P*) and *Project-Version* (*PV*) to represent software units and their specific releases, respectively. Table I shows how these terms correspond to identifiers in representative ecosystems.

2) Vulnerability Propagation Analysis: Given a vulnerability, identifying the scope and scale of its impact in software supply chains is called *vulnerability propagation analysis*, which takes vulnerability intelligence and inter-project dependencies as input to reason out the vulnerability's impact on downstream projects. Vulnerability propagation analysis can be conducted at different granularity levels. One option is the *PV*-level analysis, which considers a downstream *PV* as affected by the vulnerability if it declares a dependency on the upstream vulnerable *PV*s. The other one is the call graph (CG)-level analysis, which regards a downstream *PV* as affected only when it directly or transitively calls vulnerable functions (VFs) of the upstream vulnerable *PV*s. A prerequisite for CG-level analysis is to identify the VFs, where the vulnerable logic exists. Currently, the most widely adopted VF identification method is the patch-based approach [8], [14], [29]–[31], which identifies functions deleted or modified in patches as VFs, a widely adopted strategy due to its logical rationale and alignment with standardized patch information.

B. Limitations of Existing Solutions

1) Vulnerability Propagation Analysis: To better profile existing vulnerability propagation research, we conduct a comprehensive literature review. Prior works are mainly empirical studies on Java [4]–[18], JavaScript [19]–[23], and Python [24] ecosystems, as shown in Table II. To clearly display and compare these works, we profile them from six aspects:

Direction indicates whether the work conducts a forward analysis to answer *'which vulnerabilities in upstream dependencies affect a downstream project'*, or a backward analysis to answer *'which downstream projects (as the call sites) are affected by an upstream vulnerability—by using a chain of function calls to reach it'*. Intuitively, backward analysis is more suitable for vulnerability impact analysis, as it starts from the vulnerable site and propagates to the downstream projects.

Dep Scope clarifies whether the work examines the partial or complete dependency relations for a target ecosystem. For example, if a work only selects a subset from a software ecosystem with their dependencies for analysis, it has partial dependency scope. Instead, if the work conducts the propagation analysis with considering the whole ecosystem and dependency relations, the dependency scope is complete.

Coverage shows whether the work analyzes partial or complete projects. For example, if a work only selects one version to represent the target project, then it has partial coverage. Conversely, if the work considers all released versions for the propagation, it has complete coverage.

Transitivity indicates whether the work analyzes only the direct dependency relations or the transitive dependencies. Direct analysis only takes projects that directly depend on the vulnerable project into consideration, *i.e.,* one-hop dependency; transitive analysis considers multi-hop dependency towards the root vulnerable project.

TABLE II: Summary of related works in comparison with this work. In 'LAN', 'JA' stands for Java, 'JS' stands for JavaScript, and 'PY' stands for Python. In 'Granularity', 'PV' means the work analyzes the propagation at *PV* level, and 'CG' means CG-level analysis. In 'VF Identification', 'Manual' means the work identifies VFs manually, 'Patch' means the work uses a patch-based method to identify VFs, and 'Patch (Optimized)' means the work uses an optimized patch-based method.

Year	LAN	Research	Direction	Dep Scope	Coverage	Transitivity	Granularity	VF Identification
2015	JA	Cadariu *et al.* [4]	Forward	Partial	Partial	Direct	PV	✗
2015	JA	Ponta *et al.* [5]	Forward	Partial	Partial	Direct	PV	Patch
2017	JS	Lauinger *et al.* [19]	Forward	Partial	Partial	Transitive	PV	✗
2018	JS	Decan *et al.* [20]	Forward	Partial	Partial	Direct	PV	✗
2018	JA	Kula *et al.* [6]	Forward	Partial	Partial	Direct	PV	✗
2018	JA	Du *et al.* [7]	Forward	Partial	Partial	Direct	PV	✗
2018	JA	Ponta *et al.* [8]	Forward	Partial	Partial	Direct	CG	Patch
2018	JA	Pashchenko *et al.* [9]	Forward	Partial	Partial	Direct	PV	Patch
2019	JA	Hu *et al.* [10]	Forward	Partial	Complete	Transitive	PV	✗
2019	JS	Zimmermann *et al.* [21]	Backward	Complete	Complete	Transitive	PV	✗
2020	JA	Wang *et al.* [11]	Forward	Partial	Partial	Direct	CG	Patch
2020	JA	Ponta *et al.* [12], [32]	Forward	Partial	Partial	Direct	CG	Patch
2020	PY	Ma *et al.* [24]	Backward	Partial	Partial	Transitive	CG	Manual
2022	JS	Liu *et al.* [23]	Backward	Complete	Complete	Transitive	PV	✗
2023	JS	Wang *et al.* [22]	Backward	Complete	Complete	Transitive	PV	✗
2023	JA	Zhang *et al.* [13]	Backward	Complete	Complete	Transitive	PV	✗
2023	JA	Wu *et al.* [14]	Forward	Complete	Partial	Direct	CG	Patch
2023	JA	Mir *et al.* [15]	Forward	Partial	Complete	Transitive	CG	Patch
2024	JA	Ma *et al.* [16]	Forward	Partial	Complete	Transitive	PV	✗
2024	JA	Zhang *et al.* [17]	Backward	Partial	Complete	Direct	CG	Patch (Optimized)
2025	JA	Shen *et al.* [18]	Backward	Partial	Partial	Transitive	CG	Patch
		This Work	Backward	Complete	Complete	Transitive	CG	Patch (Optimized)

Granularity shows the granularity at which a work conducts the propagation analysis. CG-level analysis inspects whether downstream *PV*s directly or transitively call upstream VFs, which is much more accurate than *PV*-level analysis that only considers *PV* dependency relations [14].

VF identification indicates whether the work identifies VFs and how it identifies them. Manual identification cannot be scaled to large software ecosystems. Although the patch-based VF identification has been widely used in existing work, it has two limitations. First, vulnerability patches are not always publicly available [33], though several methods have been proposed to address this [33]–[40], which are beyond the scope of this paper. Second, patches sometimes contain changes unrelated to the vulnerability.

2) Vulnerability Assessment: For a software supply chain vulnerability, it is equally important to assess its own characteristics and its impact on downstream dependents. The Common Vulnerability Scoring System (CVSS) by FiRST provides a way to capture the principal characteristics of a vulnerability and produce a numerical score (0–10) reflecting its severity [25]. However, it is fundamentally designed to assess and reflect the characteristics and severity of individual vulnerabilities, and does not extend well to measuring vulnerability impacts across software supply chains. This limitation is explicitly acknowledged in the CVSS v4.0 FAQ [26], where it is clarified that *there is no prescribed way to use CVSS Base and Environmental metrics to score a vulnerability along a long supply chain*. Furthermore, prior research mainly focuses on automating existing assessments [41]–[51] or proposing new metrics [52]–[59] to profile a vulnerability's own characteristics. How to assess the impact of a vulnerability in software supply chains is still an open problem.

Overall, for vulnerability propagation analysis, an accurate and comprehensive solution should be a backward, transitive, CG-level analysis that considers complete dependency scope and project coverage and is capable of identifying VFs. Nevertheless, according to our comprehensive investigation, there still remains a gap between existing works and this goal. For vulnerability assessment, the community needs a new metric reflecting vulnerability impact in software supply chains.

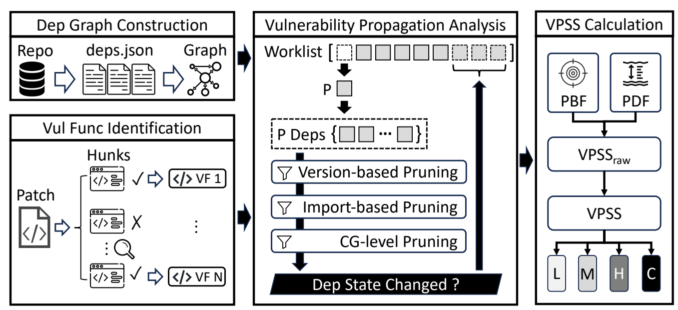


Fig. 1: Approach Overview

III. Approach

A. Overview

Figure 1 illustrates the overview of our approach, which consists of four components:

1) Dependency Graph Construction: To carry out vulnerability propagation analysis, we first need to identify all the downstream projects depending on the upstream project where the given vulnerability is located. We construct a *P*-level dependency graph for this purpose by analyzing the dependency declaration files (*e.g.,* POM files in Java Maven ecosystem) of all the projects in the target software ecosystem.

2) Vulnerable Function Identification: For CG-level analysis, VFs serve as the starting points. This component first generates a list of VF candidates using the patch-based method, and then takes a large language model (LLM)-assisted strategy to filter out vulnerability-irrelevant candidates.

3) Vulnerability Propagation Analysis: The goal of this component is to effectively identify all the downstream *PV*s in the whole ecosystem that directly or transitively call the VFs in the root upstream *PV*s where the vulnerability is located. We design a hierarchical worklist-based propagation algorithm to achieve this goal. Beginning with the root upstream *P*, each pass of the algorithm handles the direct dependencies between the upstream *P* and its downstream dependent *P*s by pruning the downstream *PV*s with hierarchical pruning methods to minimize the number of dependencies.

4) VPSS Calculation: This component is responsible for calculating the VPSS score based on the results of the propagation analysis, which considers both the breadth and depth of the vulnerability impact scope in software supply chains.

B. Dependency Graph Construction

Given a vulnerability and the *PV*s affected by it, one of the preliminaries is to figure out which *PV*s depend on these vulnerable *PV*s. The graph structure can effectively organize *PV*s and their dependency information for querying. We call this directed graph a dependency graph. Based on the level of granularity, the dependency graph can be constructed in two distinct ways: the *PV*-level and the *P*-level design. In the *PV*-level dependency graph, nodes represent *PV*s and edges denote direct dependencies between them. In contrast, the *P*-level graph abstracts nodes as *P*s, with edges summarizing relations derived from the underlying *PV*-level dependencies.

Existing graph-based studies all construct *PV*-level dependency graphs. However, based on two observations, this option is not efficient for ecosystem-scale vulnerability propagation analysis. First, modern software ecosystems have become extremely large. For example, the Java Maven ecosystem has more than 15 million *PV*s [27], and the *PV*-level dependency graph could have tens of millions of nodes and even more edges. This huge scale makes the *PV*-level dependency graph too large to be queried efficiently, especially for transitive dependencies. Second, the number of *PV*s that depend on the upstream vulnerable *PV*s is usually a minority in a target ecosystem. Even among the *PV*s that do have dependencies, the ones that are actually affected are not many [14], which means that most of the nodes and edges in the *PV*-level dependency graph are irrelevant to the vulnerability propagation analysis. Therefore, it is not necessary and inefficient to construct a *PV*-level dependency graph for a whole ecosystem.

To address these issues, we propose to construct a *P*-level dependency graph. The number of *P*s in an ecosystem is usually much smaller than that of *PV*s, meaning that the *P*-level dependency graph has a much smaller scale than the *PV*-level dependency graph. The *P*-level graph can not only reduce computational overhead, but also serve as an efficient pre-filter, allowing queries to quickly narrow down the search space.

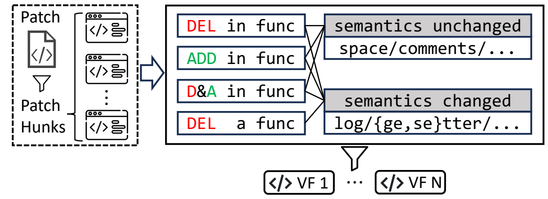


Fig. 2: Vulnerable Function Identification

With the initial traversal confined to a smaller, less complex *P*-level graph, the overall analysis becomes more scalable.

We follow a four-step procedure to build the *P*-level dependency graph. First, we download the ecosystem index and extract all the *PV* identifiers from the index into a list. Second, we obtain the dependency declaration files for *PV*s in this list from the official repository. Third, we parse the dependency declaration files to extract the dependencies of each *PV* and save them into `deps.json` files. Fourth, we construct the *P*-level dependency graph by analyzing the `deps.json` files of all the *PV*s. Specifically, for each *P*, we aggregate all the recorded dependent *PV*s from the `deps.json` files of the *PV*s that belong to it. During the propagation analysis (§ III-D), we conduct a targeted *PV*-level inspection by querying the `deps.json` files when necessary, at which point the scale of the subject has been significantly reduced.

C. Vulnerable Function Identification

As presented in Figure 2, the VF identification component consists of two steps: (1) patch-based VF candidate generation and (2) LLM-assisted VF filtering.

First, we parse the patch of the target vulnerability into individual hunks and only extract the function-modifying hunks. There are five types of function-modifying hunks: function *addition & deletion*, and internal *deletion & addition & modification (including deletion and addition)*. We filter out the function addition hunks as prior work [14] does because they are not the root cause of the vulnerability.

Second, we leverage LLMs for VF filtering. This method has three advantages: (1) LLMs possess broad domain knowledge across programming languages, enabling us to develop a language-agnostic and generalizable filtering approach. (2) LLMs are capable of recognizing non-standard syntax and syntactic sugar that are difficult to enumerate manually. (3) As LLMs continuously evolve and incorporate newly observed patterns from code corpora, their filtering capabilities remain up-to-date and adaptable, whereas manually maintained rules are often incomplete and costly to update.

Specifically, for the remaining VF hunk candidates, we design an in-context learning (ICL) [60] strategy and drive LLMs to follow two filtering principles: semantics-equivalent modification and semantics-changing modification. If the semantics of a function remain equivalent after being modified by a hunk, the hunk is considered irrelevant to the vulnerability and should be filtered out, because it does not affect the existence of the vulnerability. For example, if a hunk only changes

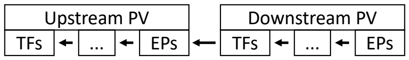


Fig. 3: Call Path Illustration

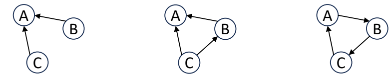
variable names, adds or deletes whitespaces, it is likely to be vulnerability-irrelevant. Even if a hunk changes the semantics of a function, it still can be irrelevant to the vulnerability. For example, if a hunk only adds or deletes logging or debugging code, it is likely to be irrelevant. To reduce incorrect filtering caused by LLMs, we also ask LLMs to provide reasons for decisions to conduct manual verification.

D. Vulnerability Propagation Analysis

The vulnerability propagation analysis is to start from a root *P* (comprising a series of *PV*s in the vulnerable version range) with a vulnerability and identify all downstream *P*s (with the corresponding *PV*s) affected by this vulnerability at CG level along the dependency graph. Specifically, given an upstream *P*, we first query the dependency graph to get its direct downstream dependent *P*s, verify the validity of dependency between each pair of upstream and downstream *P*s by inspecting whether the downstream *PV*s transitively call the VFs in the root *PV*s, and then recursively propagate the analysis only for the truly affected downstream *P*s. Figure 3 illustrates this inter-*PV* analysis: For each *PV* of an upstream *P* in affected target versions (TVs), we first need to identify the entry-point functions (EPs) that can reach the target functions (TFs) in the same upstream *PV* and be called from outside. For *PV*s of the root *P*, VFs are the TFs. Then, we need to identify which downstream *PV*s call EPs in the upstream *PV*. If any, we need to record the functions that call upstream EPs in the downstream *PV*s, which serve as the TFs in future passes.

In terms of vulnerability propagation analysis, we need to consider three possible dependency scenarios. In Figure 4a, downstream B and C have individual dependencies on A. Therefore, we can identify affected downstream *PV*s for A, B, and C sequentially (`{A,B,C}`). In Figure 4b, downstream B and C share a common dependency on A, while C also has a dependency on B. In this scenario, analysis order `{A,B,C}` and `{A,C,B}` could potentially lead to different results. The result of `{A,C,B}` is potentially incomplete, because only EPs of upstream A exist when C is being processed, and the algorithm may miss the EPs of upstream B that are called by downstream C. For `{A,C,B}`, there should be a mechanism to ensure that C is analyzed again after its upstream EPs are updated (*i.e.,* B involves new EPs). In Figure 4c, the dependencies form a cycle, which could lead to infinite analysis loops if not handled properly. Although software dependencies are expected to form a directed acyclic graph (DAG) [61], cyclic dependency relationships still exist in real-world software ecosystems. For example, in Java Maven ecosystem, *dom4j:dom4j:1.5.2* [62] and *jaxen:jaxen:1.1-beta-4* [63] have mutual dependencies on each other.

To effectively handle the aforesaid scenarios, we adopt the worklist algorithm—a well-established method in data flow



(a) (b) (c)

Fig. 4: Dependency Scenarios

frameworks—to systematically perform vulnerability propagation analysis over the dependency graph. Specifically, we maintain a worklist of *P*s whose states (*i.e.,* TVs, versioned reachable EPs, and affected downstream *PV*s) may still be updated. Initially, only the root *P* is added to the worklist. In each pass, we dequeue an item from the worklist, perform inter-*PV* analysis to update its downstream items, and enqueue the affected downstream *P*s if their associated versioned reachable upstream EPs have been updated. This approach ensures that each *P* is revisited only when necessary, effectively handling shared dependencies and cyclic structures while avoiding redundant analysis. The propagation continues until a fixed point is reached—when no new updates occur across the graph, ensuring both soundness and efficiency.

Moreover, we notice that there could be a large number of *PV* dependencies involved in the analysis, and the time and space costs associated with constructing CGs for large *PV*s are non-negligible. Consequently, it is computationally inefficient to analyze all dependencies directly at the CG level. To avoid lots of unnecessary fine-grained analysis, we employ a hierarchical pruning strategy, which first applies coarse-grained pruning methods to efficiently exclude false positive downstream dependent *PV*s (as well as their corresponding Ps, if all associated *PV*s are pruned out), and then performs the fine-grained CG-level analysis only on the remaining downstream candidates. As shown in the middle part of Figure 1, this hierarchical pruning mechanism comprises three levels of pruning: (1) *Version-based pruning*, which excludes downstream *PV*s that do not declare a dependency on the specific upstream TVs; (2) *import-based pruning*, which further eliminates downstream *PV*s that, despite declaring a dependency, do not actually import or include upstream contents; (3) *CG-level pruning*, which finally removes downstream *PV*s that do not invoke any of the upstream EPs at the CG level.

Another issue in vulnerability propagation analysis arises from the presence of fat *PV*s—release packages that bundle not only a project's own code but also its external dependencies. Such packaging practices, common in ecosystems like Java where fat JARs are widely used, can interfere with precise analysis by conflating intrinsic and extrinsic program elements. For efficient analysis, only the intrinsic scope (*i.e.,* the program components that are native to the project itself) should be considered. To address this issue, we propose a general method for identifying the intrinsic scope of a given *PV*, even when fat packaging practices differ across ecosystems. Specifically, for a target *PV*, we first obtain the set of files in its release package, denoted as *Up*. We then collect the release files of all its declared dependencies as set *Down*. By subtracting *Down*

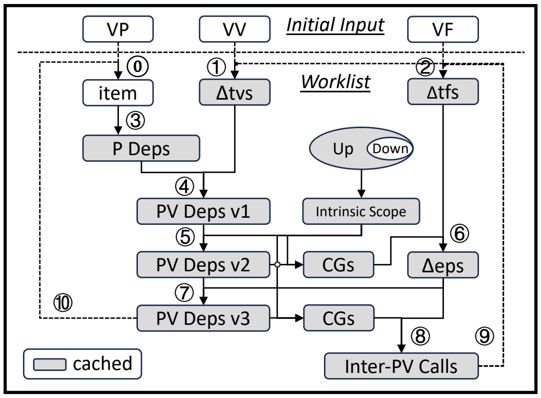


Fig. 5: Vulnerability Propagation Analysis

from *Up*, we obtain the intrinsic scope of the *PV*, which serves as the foundation for subsequent vulnerability analysis.

Taking all the above into consideration, we design a hierarchical worklist-based algorithm to perform vulnerability propagation analysis, illustrated in Figure 5, with the corresponding steps annotated using circled numbers that match those in the pseudocode provided in Algorithm 1 for clarity. Given a vulnerability, the algorithm takes three inputs: the root *P* of the vulnerability ($\mathcal{VP}$), the vulnerable versions of the root *P* ($\mathcal{VV}$), and the vulnerable functions of the root *P* ($\mathcal{VF}$). The worklist is initialized with the root *P*, and the algorithm iteratively processes items in the worklist.

At the beginning of each pass (step ⓪), one item (*P*) is fetched from the worklist. At step ①, the algorithm generates the TV list (Δtvs) of current item that are affected by the vulnerability. Specifically, if it is the first pass, the list is initialized with $\mathcal{VV}$. Otherwise, the current item must have its own upstream *P*s, the list of which is saved and updated in the previous passes (step ⑨). Consequently, the algorithm queries the *Inter-PV Calls* record of each upstream *P* in this list to get the latest affected versions of the current item into tvs. It then loads the cached old version list (old_tvs) if the current *P* has been processed in previous passes, and derives the difference between the two lists to get the new affected versions (Δtvs). The combination of old_tvs and tvs is cached for future use. At step ②, the algorithm generates the TF list (Δtfs) of the current item that are affected by the vulnerability. The generation process is similar to the generation of Δtvs. Also, the combination of old_tfs and tfs is cached for future use. Then, at step ③, the algorithm queries the dependency graph to extract the dependency relationships between the current item and its direct downstream dependent *P*s into $pdeps$.

With $pdeps$, Δtvs, and Δtfs available, the hierarchical pruning begins. At step ④, the algorithm generates the *PV* dependencies for the current item (upstream *P*) by querying the `deps.json` records belonging to *P*s in $pdeps$, and then prunes out irrelevant *PV* dependencies by checking whether the downstream *PV*s rely on upstream *PV*s covered by Δtvs. After this step, the dependency relationships between upstream *PV*s and the remaining downstream *PV*s are used to generate

Algorithm 1: Worklist-based Propagation Algorithm

Input: Vulnerable P $\mathcal{VP}$, vulnerable versions $\mathcal{VV}$, vulnerable functions $\mathcal{VF}$

Output: All the cached analysis results in Figure 5

ROOT ← **true**, worklist ← $\{\mathcal{VP}\}$
while worklist *is not empty* **do**
    ⓪ $item$ ← worklist.pop()
    (tvs, tfs) ← getTVAndTF($item$, $\mathcal{VV}$, $\mathcal{VF}$, ROOT)
    ROOT ← **false**
    (old_tvs, old_tfs) ← loadOldTVAndTF($item$)
    ① Δtvs ← diffMergeSave(old_tvs, tvs)
    ② Δtfs ← diffMergeSave(old_tfs, tfs)
    ③ $pdeps$ ← genPDeps($item$)
    // hierarchical pruning: ④ – ⑧
    $(changed, v3)$ ← prune($item$, $pdeps$, Δtvs, Δtfs)
    if $changed$ **then**
        ⑨ propagateTVAndTF($item$, $v3$)
        ⑩ worklist.extend($v3$)

or update *PV Deps v1*. At step ⑤, the algorithm further prunes the dependencies by checking whether the downstream *PV*s import contents from upstream *PV*s in *PV Deps v1*. The check is restricted to the intrinsic scope for both upstream and downstream *PV*s. After this step, *PV Deps v2* is generated or updated. Then, at step ⑥, the algorithm identifies new EPs (Δeps) for CG-level pruning. To achieve this, the algorithm loads the cached old EP list, derives the new EPs list by constructing CG and conducting backward BFS traversal from Δtfs to externally accessible functions, and uses the difference between the two lists as Δeps for each upstream *PV*. The combination of old_eps and eps is cached for future use. Notably, empty Δeps for all the upstream *PV*s indicates that the dependency state of the current item has not changed, and the algorithm will not append its downstream *P*s to the worklist. With Δeps available, at step ⑦, the algorithm prunes the dependencies by checking whether downstream *PV*s call EPs in Δeps. After this step, *PV Deps v3* ($v3$ in Algorithm 1 for brevity) is generated or updated. At the end of the pruning process, the algorithm generates and caches the *Inter-PV Calls* record for the current item at step ⑧.

At step ⑨, the algorithm sends the inter-*PV* information (versions and calls) to each downstream *P* from *PV Deps v3* for future passes. Finally, at step ⑩, the algorithm appends the downstream *P*s from *PV Deps v3* to the worklist. The algorithm continues until the worklist becomes empty.

E. VPSS Calculation

After the propagation analysis, we obtain graph-based statistics that reflect a vulnerability's impact across the ecosystem. However, these raw data are not readily interpretable or actionable. To better profile this impact, we introduce the Vulnerability Propagation Scoring System (VPSS)—a metric that transforms propagation data into a standardized and meaningful impact score for software supply chain vulnerabilities.

The design of VPSS follows three key principles: (1) Graph awareness: The metric should capture both the breadth (number of affected downstream projects) and depth (propagation chains) of a vulnerability in the dependency graph, reflecting how widely and deeply it spreads. (2) Interpretability and compatibility: VPSS must be easy to understand, ensuring seamless integration into current vulnerability management workflows and complementing static severity metrics such as CVSS. (3) Time-awareness: As ecosystems evolve, the metric should adapt to changes such as new dependencies or patches. VPSS is thus a dynamic score, supporting longitudinal tracking and timely risk assessment.

To apply these principles, VPSS transforms the results of vulnerability propagation analysis into a normalized impact score within the 0–10 range. As shown in Figure 1, the score is divided into four tiers—*low* (0–4), *medium* (4–7), *high* (7–9), and *critical* (9–10)—for intuitive risk interpretation.

VPSS captures both the scale and complexity of vulnerability propagation through two multiplicative components: the *Propagation Breadth Factor* (PBF), which quantifies how widely a vulnerability spreads via direct and transitive downstream dependencies, and the *Propagation Depth Factor* (PDF), which measures how deeply it penetrates the dependency graph based on propagation chain length. These two factors define the raw score:

$$VPSS_{\text{raw}} = PBF \times PDF \tag{1}$$

The PBF component is computed from four normalized ratios representing the proportion of affected downstream *P* and *PV* entities, separately for direct and transitive dependencies. Here, Total_P and Total_PV denote the total number of *P*s and *PV*s in the target ecosystem, respectively, which are obtained in the construction process of the dependency graph.

$$r_{\text{p\_dir}} = \frac{P_{\text{dir}}}{\text{Total\_P}}, \quad r_{\text{p\_trans}} = \frac{P_{\text{trans}}}{\text{Total\_P}},$$
$$r_{\text{pv\_dir}} = \frac{PV_{\text{dir}}}{\text{Total\_PV}}, \quad r_{\text{pv\_trans}} = \frac{PV_{\text{trans}}}{\text{Total\_PV}}$$

These values are aggregated using a weighted sum:

$$W = \begin{pmatrix} w_1 & w_2 & w_3 & w_4 \end{pmatrix}$$
$$X = \begin{pmatrix} r_{\text{p\_dir}} & r_{\text{p\_trans}} & r_{\text{pv\_dir}} & r_{\text{pv\_trans}} \end{pmatrix}$$

Generally, the relationship between these weights should be $w_1 > w_3 > w_2 > w_4$, which is based on the following considerations: Direct dependencies face greater risks than transitive dependencies; *P* dependencies are more stable and reflect their impact on the entire ecosystem, while *PV* dependencies may be less stable due to version fluctuations.

To avoid concentration of PBF values in a narrow range, we apply a logarithmic scaling with an amplification factor γ:

$$PBF = \ln\left(1 + \gamma \cdot WX^{\top}\right) \tag{2}$$

The PDF component is more straightforward, measuring the average and maximum depth of propagation paths on the dependency graph, where L_{norm} is a normalization constant used to adjust the depth metric to a reasonable scale:

$$PDF = 1 + \frac{L_{\max} + L_{\text{avg}}}{2L_{\text{norm}}} \tag{3}$$

The raw VPSS score is then normalized to the final 0–10 range using an exponential saturation function, where k is the saturation parameter controlling the rate at which the raw scores are converted to the final scores:

$$VPSS = 10 \times \left(1 - \exp\left(-\frac{VPSS_{\text{raw}}}{k}\right)\right) \tag{4}$$

In total, VPSS introduces seven parameters—w_1, w_2, w_3, w_4, γ, L_{norm}, and k—whose values affect the scaling and sensitivity of the score. Currently, we set these parameters based on domain knowledge and empirical tuning. We leave automating their setting based on statistical learning from historical vulnerability data as future work.

Lastly, to reflect the evolving nature of software ecosystems, VPSS is explicitly time-aware. As new software versions are released and patches are applied, the downstream impact of a vulnerability naturally diminishes. Particularly, when calculating a time-aware VPSS score at t, all *P*s and *PV*s released later than t will be excluded. Therefore, each VPSS score corresponds to a specific snapshot in time and should be represented in the form: `<CVE, VPSS, Timestamp>`.

IV. Implementation

We implement a prototype of our approach for the Java Maven ecosystem in 2.3K lines of Python and 1K lines of Java code. In this section, we present the implementation details.

Dependency Graph Construction. We download the Maven repository index from the Maven Central Repository (MCR) [64], and parse it with Apache Lucene [65] to extract the *PV* information. Then, we download the POM files for *PV*s in the index, and use Maven Model Builder [66] to extract the dependency information from them. We follow prior work [9] to filter out non-deployed dependencies whose `scope` are not `compile` or `runtime`. Finally, we build the *P*-level dependency graph with Python NetworkX [67], and store it in a Neo4j graph database [68] for efficient querying.

Vulnerable Function Identification. For the filtering process, we use the GPT-4o-mini API provided by OpenAI [69] to filter out the vulnerability-irrelevant VF candidates.

Vulnerability Propagation Analysis. We query the dependency graph stored in the Neo4j database to obtain all the downstream dependencies of specific *P*. To verify whether a downstream *PV* imports an upstream *PV*, we use the Java Dependency Analysis Tool (jdeps) tool provided by the OpenJDK [70] to analyze the JAR files. To quickly determine whether an upstream method is called by in a downstream *PV*, we utilize the ASM [71] to parse and search in the bytecode of the JAR files. For CG-level analysis, we use Soot [72] to analyze the bytecode of JAR files and generate call graphs.

CVE-2021-43795

```
@@ -182,7 +198,7 @@ public String toString() {
         if (query == null) {
             return path;
         }
-        return path + "?" + query;
+        return path + '?' + query;
     }

     @Nullable
```

CVE-2021-26118

```
@@ -711,14 +712,6 @@ private void delayedStop(final int wait...
         }
     }
-    public void setAdvisorySession(AMQSession amqSession) {
-        this.advisorySession = amqSession;
-    }
-
-    public AMQSession getAdvisorySession() {
-        return this.advisorySession;
-    }
```

Fig. 6: Excerpts of Patches for CVE-2021-{43795,26118}

V. Evaluation

In this section, we evaluate the effectiveness of our approach by answering the following research questions (RQs):

RQ1: How effective and scalable is our ecosystem-scale vulnerability propagation analysis in identifying potentially affected downstream software projects? (§ V-B)

RQ2: What statistical insights can be drawn from the VPSS scores computed across real-world vulnerabilities? (§ V-C)

All the experiments are conducted on a server equipped with an AMD EPYC 9184X 16-Core Processor and 500 GB of physical memory, running Ubuntu 22.04.5 LTS on the host.

A. Dataset Preparation

To evaluate the effectiveness of our full-ecosystem vulnerability propagation analysis, we build upon the dataset released by Wu *et al.* [73], which contains over 800 vulnerabilities in Maven. However, running our full analysis on all 800+ vulnerabilities would impose a heavy burden on MCR and significantly increase computational cost. To balance evaluation thoroughness and practical feasibility, we randomly sample 100 vulnerabilities as our experimental dataset.

With this dataset, we make the following enhancements. First, in Wu *et al.*'s dataset, only one vulnerable version was selected for each CVE, which can not fully capture the affected version range. To address this limitation, we enhance the dataset by incorporating the complete list of vulnerable versions for each CVE from the National Vulnerability Database (NVD) [74]. Second, we identify inaccurate VF annotations in the original dataset with our method in § III-C, and remove two incorrect VFs. Figure 6 shows excerpts from the corresponding patches [75], [76]. In CVE-2021-43795 [77], the only change is replacing double quotes with single quotes around the character '?', without altering statement semantics. Thus, the `toString()` method should not be considered a VF. In CVE-2021-26118 [78], the removed *Setter* and *Getter* methods for `AMQSession` only access `this.advisorySession` and contain no vulnerability-related logic.

In summary, each entry in our refined dataset contains the groupId and artifactId of the vulnerable project, an augmented list of vulnerable versions, and a set of vulnerable functions.

B. Vulnerability Propagation Analysis

For each vulnerability in the dataset, we run our prototype to assess its software supply chain impact in the Java Maven ecosystem. We use the snapshot of MCR index on December 26, 2024, to construct the dependency graph. There are around 660K *P*s in the dependency graph, much fewer than the 15M *PV*s in MCR. During the analyzing process, we limit the request frequency when downloading JAR files to avoid excessive load on MCR. Excluding the time spent on downloading the JAR packages, analyses for all 100 vulnerabilities are completed within one week. The completion itself demonstrates that our approach is capable of vulnerability impact assessment in a large-scale software ecosystem such as Maven.

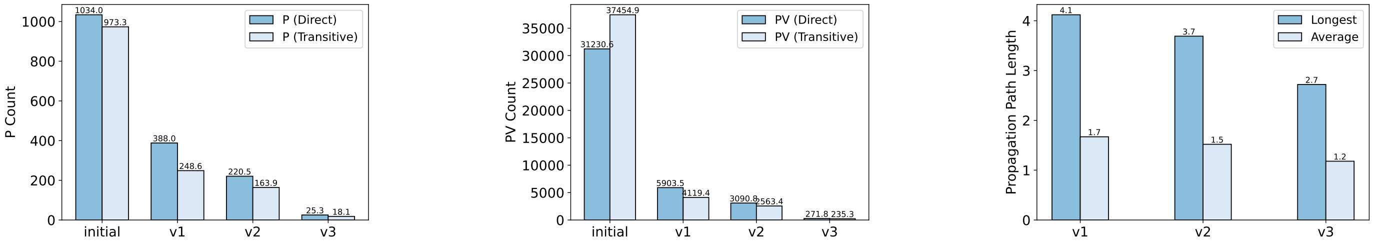
To further evaluate the effectiveness of our hierarchical worklist-based propagation algorithm, we collect data on the number of *P*s and *PV*s involved in vulnerability propagation at different stages of the algorithm, as well as the length of the longest propagation paths and the average length of propagation paths on the dependency graph. Figure 7 presents the average values of these data at different stages. For direct and transitive *P* (Figure 7a) and *PV* dependencies (Figure 7b), we obtain the average values before the whole pruning process and after each pruning step. For dependency paths (Figure 7c), we find that getting the average and longest path length before pruning would take too long to compute, so we decide to only look at the results after each pruning step.

We obtain the following findings from the analysis results: First, the hierarchical pruning mechanism is quite effective, as 97.8% *P*s and 99.2% *PV*s on average are pruned out during the vulnerability propagation analysis. Also, the length of the longest path and average length decrease at least by 34.1% and 29.4%, respectively. In addition, all the statistics in Figure 7 decrease in stages, confirming that every pruning process makes its own contributions. Considering the time and space cost of constructing CGs, our approach greatly reduces the number of CGs that need to be built. Second, performing vulnerability propagation analysis at the *PV* level based only on project-declared dependencies will result in a large number of false positives, as 94.9% *PV*s are pruned out with import-based pruning and CG-level pruning.

C. VPSS Statistics

After the propagation analysis is completed for vulnerabilities in our dataset, we collect results for VPSS calculation. As mentioned in § III-E, we determine the parameter values based on preliminary experiments and expert knowledge, aiming to balance the influence of different components and ensure that VPSS scores meaningfully reflect the propagation impact—higher scores correspond to wider and deeper impact in supply chains. Particularly, we empirically set the parameters for VPSS computation as follows: $w_1 = 5$, $w_2 = 2.5$, $w_3 = 3$, and $w_4 = 1.5$, $\gamma = 500$, $L_{\text{norm}} = 10$, and $k = 0.5$.

With such settings, for each vulnerability, we sample 24 time points at 30-day intervals starting from its disclosure date in NVD (denoted as t_0). We compute the VPSS score at each point (t_0 to t_{23}) to capture the evolution of the vulnerability's impact on software supply chains over approximately 24 months. To evaluate whether VPSS effectively captures the temporal and distributional characteristics of vulnerability propagation, we visualize its evolution and overall trends. Specifically, we present Figure 8 to highlight VPSS trajectories



(a) Average P (Direct vs Transitive) (b) Average PV (Direct vs Transitive) (c) Longest and Average Path Length

Fig. 7: Average propagation statistics across pruning stages. The v1, v2, and v3 are associated with the pruning results *PV Deps v1*, *PV Deps v2*, and *PV Deps v3* in § III-D, respectively.

of the top-10 most impactful CVEs, and Figure 9 to show the distribution of VPSS scores across all 100 CVEs over time.

As shown in Figure 8, at t_0, the top-10 VPSS scores scale from 7.35 to 4.68, reflecting risk levels from *high* to *medium*. Nevertheless, the VPSS scores generally decline over time, indicating a decreasing impact on software supply chains. This trend can be attributed to two main factors: (1) as patched versions are released, an increasing number of downstream projects migrate to vulnerability-free versions; and (2) over time, more projects emerge in the Maven ecosystem. Additionally, we observe an unusual increase in the VPSS score of CVE-2016-3086 [28] from t_0 to t_1 (from 6.95 to 7.19). This anomaly may be explained by delayed dependency updates in downstream projects [14]. Lastly, under the current parameter settings, none of the vulnerabilities in our dataset reach the *critical* VPSS risk level, leaving room for potentially higher-impact vulnerabilities beyond the scope of the current dataset.

Similar to the top-10 CVEs, in Figure 9, the distribution of VPSS scores across all 100 CVEs shows a gradual decline in both median and interquartile range over time. The boxplots reveal that the majority in the dataset maintain relatively low VPSS scores. Over time, the overall dispersion narrows slightly, suggesting that the propagation effects of most vulnerabilities tend to stabilize or diminish within two years. This aligns with the expected lifecycle of patch adoption and ecosystem decoupling from vulnerable packages.

D. Case Study: CVE-2016-5393

In this section, we take CVE-2016-5393 [79] as an example to illustrate how our framework captures vulnerability propagation in software supply chains. CVE-2016-5393 is a high-severity vulnerability (CVSS score: 8.8) in *org.apache.hadoop:hadoop-common*. In affected versions of Apache Hadoop, a remote attacker can potentially execute commands with the privileges of the HDFS service. The VFs include several command execution utilities in the `Shell` class that are widely reused by other components in Hadoop.

Our analysis shows that CVE-2016-5393 exhibits substantial propagation at the ecosystem level. At t_0, the vulnerability directly affected 228 Ps and transitively propagated to 154 others, impacting a total of 618 direct and 321 transitive PVs. The resulting VPSS score reached 7.35—the highest among all CVEs in our dataset. Over the course of 24 months, the VPSS score exhibited a gradual decline to 6.86, indicating a slow mitigation pace, which reflects long-tail dependency retention in real-world ecosystems. Notably, the longest propagation chain consists of 7 dependency hops, spanning critical components of the Hadoop and Hive data processing stacks, illustrating how a single low-level vulnerability can affect a wide range of downstream analytical and database components. The average path length also increased slightly over time, reaching 2.33 by t_{23}, indicating deepening propagation.

VI. Discussion

A. Accuracy of Propagation Analysis

Several factors affect the accuracy of our vulnerability propagation analysis. First, vulnerable versions from public databases such as NVD may be incomplete or inaccurate [80], leading to false positives or negatives in propagation results. Recent work [81]–[83] proposes version identification methods that could be integrated into our framework. Second, although we enhance patch-based VF identification, real-world security patches occasionally include unrelated but substantial code changes, complicating accurate VF extraction. These cases require a deeper understanding of vulnerability root causes and control-flow analysis to avoid false positives. We leave more precise VF identification as future work. Third, to achieve ecosystem-scale analysis, we rely on static techniques such as import analysis and CG construction. While efficient and broadly applicable, static analysis may miss paths involving dynamic features like dynamic class loading or reflection, causing false negatives. Nevertheless, our modular framework allows static analysis advances to be easily incorporated, improving accuracy without altering the core propagation logic.

B. Parameter Setting of VPSS

In designing the VPSS framework (§ III-E), we introduce a set of configurable parameters to enhance its flexibility and adaptability across different analytical contexts. In the evaluation, we instantiate these parameters with fixed values based on domain expertise (§ V-C). These settings are explicitly provided to ensure the reproducibility of our results. While this manual configuration suffices for our work, a promising direction for future work is to explore data-driven approaches for parameter tuning. For example, one could employ statistical optimization or learn optimal settings from historical

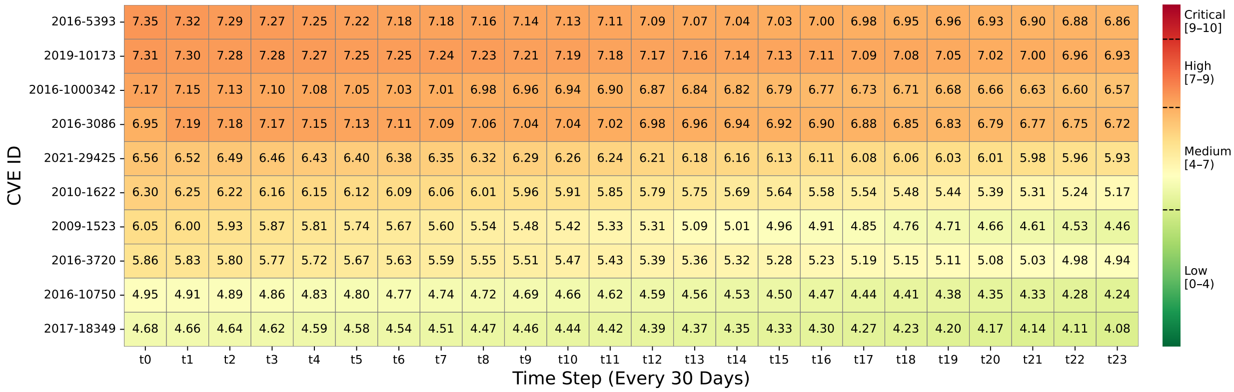


Fig. 8: VPSS Time Series (Every 30 Days) for Top-10 CVEs (t0 to t23)

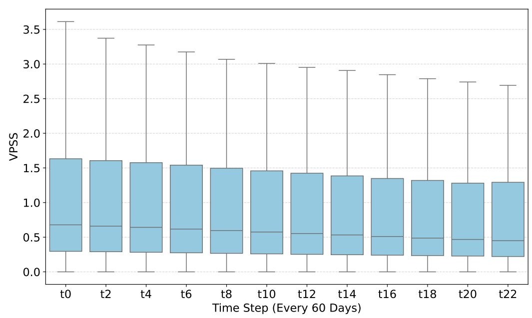


Fig. 9: VPSS Distribution (Every 60 Days) Across 100 CVEs

vulnerability propagation data, enabling more adaptive and context-aware scoring across diverse software ecosystems.

C. Application of VPSS

The VPSS framework offers three key applications. First, after a vulnerability is disclosed, VPSS enables quantification of its impact across software supply chains. It can be used with CVSS to provide more actionable and early-warning signals. Second, VPSS can be integrated into vulnerability management workflows to enhance the prioritization process, ensuring that remediation efforts focus on weaknesses with the greatest propagation risk. Finally, by translating the complex software interdependencies into a standardized score, VPSS facilitates the quantification of software supply chain risk for cyber-insurance underwriting [84], supporting more granular and data-driven policy design and premium calculation.

VII. Related Work

A. Vulnerability Propagation Analysis

Existing work on vulnerability propagation spans Java, JavaScript, and Python ecosystems: In Java, Wu *et al.* [14] and Mir *et al.* [15] perform CG-level reachability analyses on global and subset Maven graphs; Ponta *et al.* [5], [8], [9], [12], [32] combine static and dynamic methods for application-level VF detection; Zhang *et al.* [17] determine whether a given project is threatened by vulnerabilities by establishing and querying a vulnerable API database; others parse POM files for one-hop dependency analyses [4], [6], [7] or build dependency graphs for direct and transitive analyses [10], [11], [13], [16], [18]. In JavaScript, empirical studies trace client-side library usage and vulnerability inclusions [19], npm direct dependencies [20], and ecosystem-wide propagation via dependency graphs [21]–[23]. In Python, Ma *et al.* [24] propose a two-stage impact estimation for scientific projects.

B. Vulnerability Assessment

In recent years, researchers have proposed several approaches to improving existing assessments and conducting novel assessments [85]. Among them, some works aim to automatically predict the CVSS scores for vulnerabilities [41]–[46]; some propose methods to automate the Common Weakness Enumeration (CWE) classification task [47]–[51]. In addition to the widely adopted assessments, to better understand and profile vulnerabilities, researchers have begun to study more characteristics of them. One active area is exploitation prediction [52]–[59], which adopts data-driven techniques to estimate the likelihood that a vulnerability will be exploited in the wild. For instance, the Exploit Prediction Scoring System (EPSS) managed by FiRST provides probability scores between 0 and 1 for vulnerabilities as a daily estimate of exploitation being observed over the next 30 days [57].

VIII. Conclusion

In this paper, we fill two key gaps in software supply chain security: the lack of accurate whole-ecosystem vulnerability propagation analysis and the absence of quantitative indicators for assessing vulnerability propagation impact. We propose a novel framework that combines a hierarchical worklist-based algorithm with multi-level pruning to enable scalable, call-graph-level propagation analysis across direct and transitive dependencies. To quantify propagation impact, we introduce the *Vulnerability Propagation Scoring System* (VPSS), a graph-based metric capturing both propagation breadth and depth over time. We implement our approach for the Java Maven ecosystem and validate it on real-world CVEs, demonstrating the effectiveness of our analysis and the expressiveness of VPSS in assessing supply chain risk.

References

[1] MITRE, "CVE - CVE-2014-0160," https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2014-0160, 2014.

[2] ——, "CVE - CVE-2021-44228," https://cve.mitre.org/cgi-bin/cvename.cgi?name=cve-2021-44228, 2021.

[3] CSA, "The Log4Shell Vulnerability," https://www.csa.gov.sg/resources/publications/the-log4shell-vulnerability, 2022.

[4] M. Cadariu, E. Bouwers, J. Visser, and A. Van Deursen, "Tracking known security vulnerabilities in proprietary software systems," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 2015, pp. 516–519.

[5] H. Plate, S. E. Ponta, and A. Sabetta, "Impact assessment for vulnerabilities in open-source software libraries," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2015, pp. 411–420.

[6] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies? an empirical study on the impact of security advisories on library migration," *Empirical Software Engineering*, vol. 23, pp. 384–417, 2018.

[7] D. Du, X. Ren, Y. Wu, J. Chen, W. Ye, J. Sun, X. Xi, Q. Gao, and S. Zhang, "Refining traceability links between vulnerability and software component in a vulnerability knowledge graph," in *Web Engineering: 18th International Conference, ICWE 2018, Cáceres, Spain, June 5-8, 2018, Proceedings 18*. Springer, 2018, pp. 33–49.

[8] S. E. Ponta, H. Plate, and A. Sabetta, "Beyond metadata: Code-centric and usage-based analysis of known vulnerabilities in open-source software," in *2018 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2018, pp. 449–460.

[9] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vulnerable open source dependencies: Counting those that matter," in *Proceedings of the 12th ACM/IEEE international symposium on empirical software engineering and measurement*, 2018, pp. 1–10.

[10] W. Hu, Y. Wang, X. Liu, J. Sun, Q. Gao, and Y. Huang, "Open source software vulnerability propagation analysis algorithm based on knowledge graph," in *2019 IEEE International Conference on Smart Cloud (SmartCloud)*. IEEE, 2019, pp. 121–127.

[11] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu, "An empirical study of usages, updates and risks of third-party libraries in java projects," in *2020 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2020, pp. 35–45.

[12] S. E. Ponta, H. Plate, and A. Sabetta, "Detection, assessment and mitigation of vulnerabilities in open source dependencies," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3175–3215, 2020.

[13] L. Zhang, C. Liu, S. Chen, Z. Xu, L. Fan, L. Zhao, Y. Zhang, and Y. Liu, "Mitigating persistence of open-source vulnerabilities in maven ecosystem," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 191–203.

[14] Y. Wu, Z. Yu, M. Wen, Q. Li, D. Zou, and H. Jin, "Understanding the threats of upstream vulnerabilities to downstream projects in the maven ecosystem," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 1046–1058.

[15] A. M. Mir, M. Keshani, and S. Proksch, "On the effect of transitivity and granularity on vulnerability propagation in the maven ecosystem," in *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023, pp. 201–211.

[16] Z. Ma, S. Mondal, T.-H. Chen, H. Zhang, and A. E. Hassan, "Vulnet: Towards improving vulnerability management in the maven ecosystem," *Empirical Software Engineering*, vol. 29, no. 4, p. 83, 2024.

[17] F. Zhang, L. Fan, S. Chen, M. Cai, S. Xu, and L. Zhao, "Does the vulnerability threaten our projects? automated vulnerable api detection for third-party libraries," *IEEE Transactions on Software Engineering*, 2024.

[18] Y. Shen, X. Gao, H. Sun, and Y. Guo, "Understanding vulnerabilities in software supply chains," *Empirical Software Engineering*, vol. 30, no. 1, pp. 1–38, 2025.

[19] T. Lauinger, A. Chaabane, S. Arshad, W. Robertson, C. Wilson, and E. Kirda, "Thou shalt not depend on me: Analysing the use of outdated javascript libraries on the web," *arXiv preprint arXiv:1811.00918*, 2018.

[20] A. Decan, T. Mens, and E. Constantinou, "On the impact of security vulnerabilities in the npm package dependency network," in *Proceedings of the 15th international conference on mining software repositories*, 2018, pp. 181–191.

[21] M. Zimmermann, C.-A. Staicu, C. Tenny, and M. Pradel, "Small world with high risks: A study of security threats in the npm ecosystem," in *28th USENIX Security symposium (USENIX security 19)*, 2019, pp. 995–1010.

[22] Y. Wang, P. Sun, L. Pei, Y. Yu, C. Xu, S.-C. Cheung, H. Yu, and Z. Zhu, "Plumber: Boosting the propagation of vulnerability fixes in the npm ecosystem," *IEEE Transactions on Software Engineering*, vol. 49, no. 5, pp. 3155–3181, 2023.

[23] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, and X. Peng, "Demystifying the vulnerability propagation and its evolution via dependency trees in the npm ecosystem," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 672–684.

[24] W. Ma, L. Chen, X. Zhang, Y. Feng, Z. Xu, Z. Chen, Y. Zhou, and B. Xu, "Impact analysis of cross-project bugs on software ecosystems," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 100–111.

[25] FiRST, "Common Vulnerability Scoring System," https://www.first.org/cvss/, 2025.

[26] ——, "CVSS v4.0 Frequently Asked Questions," https://www.first.org/cvss/v4.0/faq, 2025.

[27] MVN, "Maven Repository: Central," https://mvnrepository.com/repos/central, 2025.

[28] NVD, "NVD - CVE-2016-3086," https://nvd.nist.gov/vuln/detail/CVE-2016-3086, 2017, [Accessed 25-05-2025].

[29] J. Dai, Y. Zhang, Z. Jiang, Y. Zhou, J. Chen, X. Xing, X. Zhang, X. Tan, M. Yang, and Z. Yang, "{BScout}: Direct whole patch presence test for java executables," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1147–1164.

[30] Z. Jiang, Y. Zhang, J. Xu, Q. Wen, Z. Wang, X. Zhang, X. Xing, M. Yang, and Z. Yang, "Pdiff: Semantic-based patch presence testing for downstream kernels," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, 2020, pp. 1149–1163.

[31] B. Liu, G. Meng, W. Zou, Q. Gong, F. Li, M. Lin, D. Sun, W. Huo, and C. Zhang, "A large-scale empirical study on vulnerability distribution within projects and the lessons learned," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 1547–1559.

[32] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vuln4real: A methodology for counting actually vulnerable dependencies," *IEEE Transactions on Software Engineering*, vol. 48, no. 5, pp. 1592–1609, 2020.

[33] C. Xu, B. Chen, C. Lu, K. Huang, X. Peng, and Y. Liu, "Tracking patches for open source software vulnerabilities," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2022, pp. 860–871.

[34] Y. Zhou, J. K. Siow, C. Wang, S. Liu, and Y. Liu, "Spi: Automated identification of security patches via commits," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 1, pp. 1–27, 2021.

[35] X. Tan, Y. Zhang, C. Mi, J. Cao, K. Sun, Y. Lin, and M. Yang, "Locating the security patches for disclosed oss vulnerabilities with vulnerability-commit correlation ranking," in *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021, pp. 3282–3299.

[36] S. Wang, Y. Zhang, L. Bao, X. Xia, and M. Wu, "Vcmatch: a ranking-based approach for automatic security patches localization for oss vulnerabilities," in *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2022, pp. 589–600.

[37] B. Wu, S. Liu, R. Feng, X. Xie, J. Siow, and S.-W. Lin, "Enhancing security patch identification by capturing structures in commits," *IEEE Transactions on Dependable and Secure Computing*, 2022.

[38] T. Dunlap, E. Lin, W. Enck, and B. Reaves, "Vfcfinder: Pairing security advisories and patches," in *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, 2024, pp. 1128–1142.

[39] S. Scalco and R. Paramitha, "Hash4patch: A lightweight low false positive tool for finding vulnerability patch commits," in *2024 IEEE/ACM 21st International Conference on Mining Software Repositories (MSR)*. IEEE, 2024, pp. 733–737.

[40] J. Yu, Y. Chen, D. Tang, X. Liu, X. Wang, C. Wu, and H. Tang, "Llm-enhanced software patch localization," *arXiv preprint arXiv:2409.06816*, 2024.

[41] Z. Han, X. Li, Z. Xing, H. Liu, and Z. Feng, "Learning to predict severity of software vulnerability using only vulnerability description," in *2017 IEEE International conference on software maintenance and evolution (ICSME)*. IEEE, 2017, pp. 125–136.

[42] C. Elbaz, L. Rilling, and C. Morin, "Fighting n-day vulnerabilities with automated cvss vector prediction at disclosure," in *Proceedings of the 15th International Conference on Availability, Reliability and Security*, 2020, pp. 1–10.

[43] T. H. M. Le, D. Hin, R. Croft, and M. A. Babar, "Deepcva: Automated commit-level vulnerability assessment with deep multi-task learning," in *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2021, pp. 717–729.

[44] T. H. M. Le and M. A. Babar, "On the use of fine-grained vulnerable code statements for software vulnerability assessment models," in *Proceedings of the 19th International Conference on Mining Software Repositories*, 2022, pp. 621–633.

[45] E. Aghaei, E. Al-Shaer, W. Shadid, and X. Niu, "Automated cve analysis for threat prioritization and impact prediction," *arXiv preprint arXiv:2309.03040*, 2023.

[46] S. Pan, L. Bao, J. Zhou, X. Hu, X. Xia, and S. Li, "Towards more practical automation of vulnerability assessment," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.

[47] S. Pan, L. Bao, X. Xia, D. Lo, and S. Li, "Fine-grained commit-level vulnerability type prediction by cwe tree structure," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2023, pp. 957–969.

[48] M. Fu, V. Nguyen, C. K. Tantithamthavorn, T. Le, and D. Phung, "Vulexplainer: A transformer-based hierarchical distillation for explaining vulnerability types," *IEEE Transactions on Software Engineering*, vol. 49, no. 10, pp. 4550–4565, 2023.

[49] X.-C. Wen, C. Gao, F. Luo, H. Wang, G. Li, and Q. Liao, "Livable: exploring long-tailed classification of software vulnerability types," *IEEE Transactions on Software Engineering*, 2024.

[50] Y. Luo, W. Xu, and D. Xu, "Predicting code vulnerability types via heterogeneous gnn learning," in *European Symposium on Research in Computer Security*. Springer, 2024, pp. 67–87.

[51] C. Ji, S. Yang, H. Sun, and Y. Zhang, "Applying contrastive learning to code vulnerability type classification," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, 2024, pp. 11 942–11 952.

[52] M. Bozorgi, L. K. Saul, S. Savage, and G. M. Voelker, "Beyond heuristics: learning to classify vulnerabilities and predict exploits," in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, 2010, pp. 105–114.

[53] C. Sabottke, O. Suciu, and T. Dumitraș, "Vulnerability disclosure in the age of social media: Exploiting twitter for predicting {Real-World} exploits," in *24th USENIX security symposium (USENIX security 15)*, 2015, pp. 1041–1056.

[54] N. Tavabi, P. Goyal, M. Almukaynizi, P. Shakarian, and K. Lerman, "Darkembed: Exploit prediction with neural language models," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, no. 1, 2018.

[55] H. Chen, R. Liu, N. Park, and V. Subrahmanian, "Using twitter to predict when vulnerabilities will be exploited," in *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data Mining*, 2019, pp. 3143–3152.

[56] J. Jacobs, S. Romanosky, I. Adjerid, and W. Baker, "Improving vulnerability remediation through better exploit prediction," *Journal of Cybersecurity*, vol. 6, no. 1, p. tyaa015, 2020.

[57] J. Jacobs, S. Romanosky, B. Edwards, I. Adjerid, and M. Roytman, "Exploit prediction scoring system (epss)," *Digital Threats: Research and Practice*, vol. 2, no. 3, pp. 1–17, 2021.

[58] O. Suciu, C. Nelson, Z. Lyu, T. Bao, and T. Dumitraș, "Expected exploitability: Predicting the development of functional vulnerability exploits," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 377–394.

[59] J. Jacobs, S. Romanosky, O. Suciu, B. Edwards, and A. Sarabi, "Enhancing vulnerability prioritization: Data-driven exploit predictions with community-driven insights," in *2023 IEEE European symposium on security and privacy workshops (euroS&pW)*. IEEE, 2023, pp. 194–206.

[60] Q. Dong, L. Li, D. Dai, C. Zheng, J. Ma, R. Li, H. Xia, J. Xu, Z. Wu, T. Liu *et al.*, "A survey on in-context learning," *arXiv preprint arXiv:2301.00234*, 2022.

[61] R. C. Martin, "Acyclic dependencies principle," https://en.wikipedia.org/wiki/Acyclic_dependencies_principle, 2015.

[62] Maven, "Maven Central: dom4j:dom4j:1.5.2," https://central.sonatype.com/artifact/dom4j/dom4j/1.5.2, 2005.

[63] ——, "Maven Central: jaxen:jaxen:1.1-beta-4," https://central.sonatype.com/artifact/jaxen/jaxen/1.1-beta-4, 2005.

[64] H. Boutemy, "Central Index," https://maven.apache.org/repository/central-index.html, 2025.

[65] Apache, "Welcome to Apache Lucene," https://lucene.apache.org/, 2025.

[66] H. Boutemy, "Maven model builder," https://maven.apache.org/ref/3.8.8/maven-model-builder/, 2023.

[67] NetworkX, "NetworkX," https://networkx.org/, 2014.

[68] Neo4j, "Neo4j Graph Database," https://neo4j.com/, 2025.

[69] OpenAI, "GPT-4o mini Model," https://platform.openai.com/docs/models/gpt-4o-mini, 2025.

[70] OpenJDK, "Java Dependency Analysis Tool - Java Dependency Analysis Tool - OpenJDK Wiki," https://wiki.openjdk.org/display/JDK8/Java+Dependency+Analysis+Tool, 2025.

[71] ASM, "ASM," https://asm.ow2.io/, 2025.

[72] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: A java bytecode optimization framework," in *CASCON First Decade High Impact Papers*, 2010, pp. 214–224.

[73] Y. W. et al., "MavenEcoSysResearch/csv/CVE.csv at main," https://github.com/CGCL-codes/MavenEcoSysResearch/blob/main/csv/CVE.csv, 2023.

[74] NIST, "NVD - Home," https://nvd.nist.gov/, 2024.

[75] T. Lee, "Merge pull request from GHSA-8fp4-rp6c-5gcv," https://github.com/line/armeria/commit/e2697a575e9df6692b423e02d731f293c1313284, 2021.

[76] gtully, "Merge pull request 3315," https://github.com/apache/activemq-artemis/commit/7dd50bd58ed2b46aa7ecdba7cffe979b672d58fb, 2020.

[77] NVD, "NVD - CVE-2021-43795," https://nvd.nist.gov/vuln/detail/CVE-2021-43795, 2021.

[78] ——, "NVD - CVE-2021-26118," https://nvd.nist.gov/vuln/detail/CVE-2021-26118, 2021.

[79] ——, "NVD - CVE-2016-5393 — nvd.nist.gov," https://nvd.nist.gov/vuln/detail/CVE-2016-5393, 2021, [Accessed 26-05-2025].

[80] B. Ruan, J. Liu, C. Zhang, and Z. Liang, "Kernjc: Automated vulnerable environment generation for linux kernel vulnerabilities," *arXiv preprint arXiv:2404.11107*, 2024.

[81] L. Bao, X. Xia, A. E. Hassan, and X. Yang, "V-szz: automatic identification of version ranges affected by cve vulnerabilities," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2352–2364.

[82] S. Wu, R. Wang, K. Huang, Y. Cao, W. Song, Z. Zhou, Y. Huang, B. Chen, and X. Peng, "Vision: Identifying affected library versions for open source software vulnerabilities," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 1447–1459.

[83] Y. Cheng, L. K. Shar, T. Zhang, S. Yang, C. Dong, D. Lo, S. Lv, Z. Shi, and L. Sun, "Llm-enhanced static analysis for precise identification of vulnerable oss versions," *arXiv preprint arXiv:2408.07321*, 2024.

[84] S. Dambra, L. Bilge, and D. Balzarotti, "Sok: Cyber insurance–technical challenges and a system security roadmap," in *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2020, pp. 1367–1383.

[85] B. Ruan, J. Liu, W. Zhao, and Z. Liang, "Vulzoo: A comprehensive vulnerability intelligence dataset," in *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, 2024, pp. 2334–2337.