ReachCheck: Compositional Library-Aware Call Graph Reachability Analysis in the IDEs

CHAO WANG, School of Informatics, Xiamen University, China
LI LIN, School of Informatics, Xiamen University, China
CHENGPENG WANG, Purdue University, USA
JIAFENG HUANG, School of Informatics, Xiamen University, China
CONGXIA WU, School of Informatics, Xiamen University, China
RONGXIN WU*, School of Informatics, Xiamen University, China

Call graph reachability analysis is essential for vulnerability detection, dependency conflict analysis, and compatibility checks. However, modern software systems, particularly those developed within integrated development environments (IDEs), often rely on third-party libraries (TPLs), which significantly increase the analysis cost. This paper introduces ReachCheck, a compositional library-aware analysis for method pair reachability in the IDEs. Specifically, ReachCheck summarizes TPL reachability via offline transitive closure and integrates the summaries with application code on demand, eliminating redundant analysis. Additionally, we use matrix representations for call graphs and employ fast matrix multiplication for transitive closure, further improving efficiency. We have implemented our approach as a prototype and evaluated it upon real-world projects. Compared to online traversal, function summary approaches and three state-of-the-art graph reachability approaches (Ferrari, BL and BFL), ReachCheck achieves 237.75×, 78.55×, 84.86×, 4369.09× and 80.91× speedup respectively. For downstream clients like dependency conflict detection and CVE risk detection, ReachCheck completes analysis in 0.61 and 0.35 seconds, yielding 537.59× and 519.03× speedup over existing techniques.

CCS Concepts: • **Software and its engineering → Risk management**; **Integrated and visual development environments**; **Software libraries and repositories**; **Software maintenance tools**;

Additional Key Words and Phrases: Call graph analysis, reachability analysis, third-party libraries

1 INTRODUCTION

Call graph reachability analysis, which aims to determine whether a specific method may invoke a target method directly or transitively [51], stands as a foundational program analysis problem. Its effective solution can significantly benefit various downstream clients. Within the realm of software component analysis (SCA), for instance, developers often need to pinpoint the methods within their application code that could potentially be influenced by vulnerable methods in third-party libraries (TPLs) [39, 74]. The identified reachability facts from application code methods to vulnerable TPL methods within the call graph serve to unveil potential vulnerabilities

*Corresponding Author

Authors' addresses: Chao Wang, School of Informatics, Xiamen University, Xiamen, China, wangc@stu.xmu.edu.cn; Li Lin, School of Informatics, Xiamen University, Xiamen, China, linli1210@stu.xmu.edu.cn; Chengpeng Wang, Purdue University, West Lafayette, IN, USA, wang6590@purdue.edu; Jiafeng Huang, School of Informatics, Xiamen University, Xiamen, China, thghjf@stu.xmu.edu.cn; Congxia Wu, School of Informatics, Xiamen University, Xiamen, China, wucongxia1@stu.xmu.edu.cn; Rongxin Wu, School of Informatics, Xiamen University, Xiamen, China, wurongxin@xmu.edu.cn.

ACM 1557-7392/2025/9-ART
https://doi.org/10.1145/3767166

stemming from security gaps within the software supply chain. Besides, clients engaged in other program analyses, such as dependency conflict detection [67–69], TPL compatibility analysis [66, 83], and software debloating [35], also rely heavily on call graph reachability analysis, leveraging it as an important building block to support specific applications.

Despite several decades of research on call graph reachability analysis, there has been a notable scarcity of studies focusing on its instantiation within Integrated Development Environments (IDEs). In the IDE, where developers edit code incrementally, immediate feedback on whether a newly written method may call a vulnerable method from a TPL enables proactive detection of security and quality issues at an early stage. Identifying such issues during development, rather than post-deployment, can significantly reduce the cost and effort of vulnerability remediation [79]. Therefore, we focus on incremental vulnerability detection, where developers make frequent and localized code changes. When a new method call is added or modified, the IDE should promptly determine whether this call may transitively reach a vulnerable method in the dependency tree. To support this workflow efficiently, we formulate the analysis as a series of pairwise reachability queries, each checking whether the newly added method can reach a known vulnerable method. This formulation aligns well with the on-demand, method-level feedback expected in IDEs. It avoids the overhead of whole-program reanalysis after each code change. However, achieving this objective poses a significant challenge, particularly given the stringent time constraints imposed by the IDEs, which typically necessitates finishing analysis tasks within mere milliseconds or seconds [45, 79]. Existing techniques typically construct a call graph first and then leverage Depth-First Search (DFS) or Breadth-First Search (BFS) algorithms to traverse the graph [67–69, 74]. Nonetheless, conducting such analyses on large-scale call graphs inevitably incurs significant time overhead. In our preliminary study encompassing 15 Java open-source projects, a DFS or BFS-based approach averagely spends 327.93 seconds addressing reachability queries. Hence, accelerating call graph reachability analysis holds the potential to yield substantial benefits for a diverse array of program analysis clients within the IDEs.

According to existing literature, leveraging a graph index to accelerate the graph reachability analysis is one promising solution [36, 56, 61, 65, 78, 80]. Unfortunately, the performance of such techniques would be far from satisfactory in answering call graph reachability queries in our scenarios. This is because, the effectiveness of these approaches generally rely on the assumption of steady graph structures, which is often broken in the IDEs during the development. Specifically, the addition, removal, and replacement of TPLs are common operations in IDEs, which can lead to significant changes of call graph structure. Under such circumstances, graph-indexing approaches necessitate reconstructing a graph index from scratch, resulting in substantial computational costs. To mitigate this issue, an intuitive idea is to construct an index for each TPL using an offline analysis, and then combine their indices on-demand for performing call graph reachability analysis online. Current graph indexing structures are generally designed for a single, stable graph and do not adapt to graph changes. Consequently, when an on-demand composition of indexes is required, these structures necessitate recalculating the indexes, making it impossible to reuse those previously computed for TPLs.

This work presents a novel compositional approach to achieve an efficient call graph reachability analysis in the presence of TPLs. Basically, our key idea originates from the two observations. First, although the project under development frequently changes, the code in the TPLs is stable, which permits us to reuse the reachability summaries of TPLs. Decoupling the TPLs from the projects can avoid redundantly analyzing the TPLs multiple times for different projects, which can significantly reduce the time overhead. Second, not all the TPLs are relevant to a reachability query. Only those TPLs that directly or transitively depend on the queried TPL are useful for answering the given reachability query. Therefore, opting to load solely the summaries of necessary TPLs on-demand, rather than loading the ones of all TPLs indiscriminately, will significantly reduce the computation overhead.

Based on the above insights, we propose a compositional call graph reachability analysis ReachCheck. Technically, it consists of two lines of analysis, which target the TPLs and the project, respectively. For each TPL,

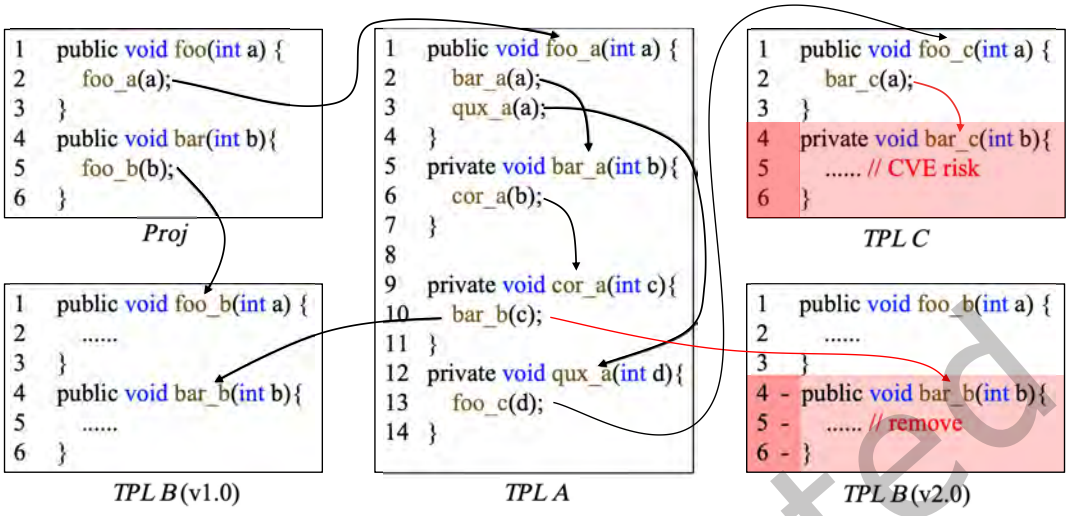

Fig. 1. A motivating code example.

ReachCheck first collects all the call edges and summarizes the reachability relation of the call graph by computing the transitive closure (TC) in an offline manner. Notably, the computed summaries only preserve the portions of the call graph that are reachable from accessible methods (See Section 4.1). By persisting the summaries related to the accessible method to the disk, ReachCheck can reuse the analysis results when analyzing the project. Second, when answering the reachability query upon the project using TPLs, ReachCheck constructs the call graph for the project and retrieves the summaries of the TPLs on demand, based on which the reachability relation can be determined. To achieve further acceleration, ReachCheck leverages the fast matrix multiplication algorithm, which can significantly improve the efficiency in the TC computation.

We implement our approach as a prototype and evaluate it using 100 popular open-source Java projects in GitHub. It is shown that ReachCheck achieves 237.75× speedup over the online traversal approach and 78.55× speedup over the function summaries approach. Meanwhile, ReachCheck achieves 84.86×, 4369.09× and 80.91× speedup over three state-of-the-art graph indexing approaches, namely Ferrari [56], BL [78] and BFL [61]. To demonstrate the usefulness of ReachCheck, we instantiate two clients of call graph reachability analysis, i.e., dependency conflict detection and CVE risk detection. Our experiments demonstrate that ReachCheck significantly enhances the performance of both applications, thereby facilitating their integration into IDEs. Specifically, ReachCheck successfully reproduces all the dependency conflicts detected by Decca [67] with the average analysis time of 0.61 seconds for each project. For CVE risk detection, ReachCheck finishes the analysis in an average of 0.35 seconds for each project and report 43 issues in total, which have been confirmed and fixed by developers. In summary, the contributions of this paper are as follows:

- We propose ReachCheck, a novel compositional library-aware call graph reachability analysis that addresses the reachability queries of method pairs in the IDE.
- To answer the reachability of a given pair of methods efficiently, we introduce a fast matrix multiplication algorithm to summarize the application and each TPL and stitch the summaries in a demand-driven manner, which incurs low time and memory overheads.
- We conduct an extensive evaluation upon real-world Java programs. The results demonstrate that our approach can significantly speedup dependency conflict detection and CVE risk detection, achieving an average 537.59× and 519.03× speedup.

2 PRELIMINARIES

2.1 Acronyms and Their Definitions

This paper introduces numerous technical and scholarly concepts. To enhance readability, we consistently use their corresponding acronyms. For easy reference, Table 1 provides a complete glossary of all acronyms along with their definitions, as well as their occurrences in the paper.

Table 1. Definitions of acronyms used in this study

	Acronym	Definition	Detail
1	SCA	Software Component Analysis	See in Section 1
2	TPL	Third-party Library	See in Section 1
3	IDE	Integrated Development Environment	See in Section 1
4	DFS	Depth-First Search	See in Section 1
5	BFS	Breadth-First Search	See in Section 1
6	TC	Transitive Closure	See in Section 1
7	OT	Online Traversal	See in Section 6.1
8	FS	Function Summary	See in Section 6.1

2.2 Motivation

Call graph reachability essentially determines whether there exists a valid call path from a source node in the user project to a target node in the TPL in the call graph. It has been widely used in various applications, including *dependency conflict detection* and *CVE risk detection*. Specifically, dependency conflict detection refers to issues that arise when different versions of TPLs in a project are incompatible, which can lead to runtime exceptions or subtle behavioral inconsistencies, due to missing methods or unexpected API changes that manifest only during execution [67]. As shown in Figure 1, the TPL *A* utilized by the client project invokes a method named `bar_b` offered by the TPL *B* v1.0. If the developers import a wrong version of TPL *B* without the method `bar_b`, for example, TPL *B* v2.0, a dependency conflict issue would arise. Obviously, such dependency conflicts can be effectively detected by analyzing call graph reachability relations between the methods in the project and TPLs. Another important downstream analysis is the CVE risk detection. As shown in Figure 1, if a method offered by a TPL, i.e., the method `bar_c` in the TPL *C*, is risky and assigned with a CVE id, we need to identify the methods in the current project invoking the risky method directly or transitively, such as the method `foo`. Similar to the dependency conflict detection, an effective call graph reachability analysis enables us to discover potential vulnerabilities caused by the usage of the risky TPL method, which can effectively promote the reliability of the projects in the presence of TPLs.

Despite decades of research into call graph reachability analysis [35, 66–69, 83], there have been few attempts that implement these techniques in IDEs to support analysis during the development phase, which is crucial for proactively addressing software security and quality issues early in industrial scenarios [45, 79]. Existing IDEs, such as IntelliJ IDEA [7], have integrated plugins Package Analyzer for detecting dependency conflicts and CVE risks [2, 8]. Regarding functionality, the IntelliJ IDEA's Package Analyzer identifies vulnerable libraries by matching the versions declared in the project's pom.xml against known entries in a vulnerability database. This approach can lead to false positives, as it does not check whether the vulnerable methods are actually invoked by the application code. Simply detecting whether an application uses a vulnerable library version may not be sufficient, as the vulnerable methods might not be reachable from the application code—a scenario commonly regarded as a false positive [38, 49, 50]. Moreover, a prior study [33] has shown that developers are often reluctant

to update dependencies due to fear of introducing breaking changes. Although several specific detectors, such as Decca [67], Riddle [68], and Insight [74], have made significant progress in reducing false positives, the unacceptable overhead introduced by call graph reachability analysis severely impede their adoption within IDE. For example, as shown in Table 2, we investigated the time spent on dependency conflict detection using Decca and Riddle on 15 open-source Java projects. The results show that performing call graph analysis together with DFS or BFS based approaches spend an average of 327.93 seconds resolving reachability queries on 198,082 call edges and 24,618 methods. To mitigate the time cost, pre-analyzing TPLs to generate call graphs for caching offline and then performing online queries by on-demand loading the cached call graphs [43] is one promising solution. The column "**TPL_CE**" in Table 2 shows the time cost of offline analyzing call graphs of the project and its TPLs using Class Hierarchy Analysis (CHA) in the open source tool Soot [4]. On average, the time cost of this offline analysis is 115.32 seconds. Despite that, the time cost of the online reachability queries on the cached call graphs is still high, on average 212.61 seconds for the same dependency conflict tasks mentioned above, which is unacceptable in IDEs. To better illustrate this cost, we include the column "**Q.**" in Table 2, which reports the number of reachability queries for each project. Each query is defined as a pair (m_s, m_t), where the m_s represents the source method in the client project from which reachability is queried, and m_t is the target method in a TPL being queried. Specifically, in Decca, m_s corresponds to a method within the client project, while m_t refers to a method that exists in a version of a TPL excluded by Maven due to dependency conflicts[1]. For instance, the `platform` project involves 7,922 queries, each verifying whether a method in the application can reach a method in a TPL. It is apparent that such high time cost does not meet the efficiency requirement of IDE plugins. Existing literature [45, 79] indicates that the analysis time of an IDE plugin should be within a few seconds or even milliseconds. Therefore, accelerating the call graph reachability analysis is critical for various program analysis client within IDEs.

2.3 Preliminaries

According to our investigation, existing studies [36, 56, 61, 65, 78, 80] on graph reachability analysis attempt to achieve the acceleration by constructing graph indices. However, the adaptation of such techniques fails to yield satisfactory performance in our scenarios. The complex and dynamic environment of an IDE, such as the addition, deletion, or replacement of a TPL, often necessitates the recalculation of index nodes, thereby incurring significant overhead. For example, Dual [65] traverses the graph and generates a spanning tree with interval-based labels. Each node in this tree is assigned an ID and an interval $[a, b)$. By comparing these intervals and IDs, reachability can be determined quickly. However, if TPLs are added, removed or replaced, it will result in the changes in the degrees of the nodes in the tree, which requires a recalculation of the IDs and intervals of the spanning tree. Repeatedly indexing process would introduce significant overhead and further degrades the efficiency of the overall reachability analysis.

An intuitive adaptation is to create graph indices offline for each TPL and dynamically combine these indices during queries to handle graph changes caused by TPL modifications. However, the existing graph indexing approaches cannot work well in such an adaptation. Specifically, indexing-based approaches typically label a series of nodes and determine reachability by checking if the labels of two nodes intersect. In essence, existing graph indexing approaches are unable to directly answer the question about which nodes are reachable from a given node. Therefore, to determine the reachability from a source node, one must perform a reachability query to build the connection between that node and all other nodes in the graph, which essentially degenerates into the BFS/DFS traversal approach. As shown in Figure 1, when combining each TPL's index, we must first query the reachability from each method in the TPL A to each method in the TPL B, since these indices alone do not

[1]When a project depends on multiple versions of the same library, Maven will exclude one of them to resolve the conflict, and the m_t is a missing method from the TPL due to dependency conflicts [67].

Table 2. The time costs in the DC detection

Projects	Sum.	TPL_CE	R_1	RA	R_2	Q.	D.
beam[9]	666.07s	303.23s	45.53%	295.19s	44.32%	25,492	30
truth[21]	63.77s	39.40s	61.78%	22.82s	35.78%	1,700	16
platform[17]	597.75s	151.14s	25.28%	414.81s	69.40%	7,922	80
webcam[23]	267.39s	101.09s	37.81%	156.61s	58.57%	1,672	52
hadoop[12]	34.88s	14.66s	42.03%	19.31s	55.36%	472	29
storm[25]	387.67s	116.92s	30.16%	249.90s	64.46%	18,400	26
saturn[22]	492.47s	111.06s	22.55%	334.50s	67.92%	62,098	43
ff4j-s[18]	1154.82s	358.64s	31.06%	782.08s	67.72%	2,082	60
azure[19]	179.63s	63.77s	35.50%	108.15s	60.21%	852	32
st-js[14]	72.78s	37.56s	51.61%	33.26s	45.70%	276	16
dubbo[15]	418.23s	122.60s	29.31%	269.88s	64.53%	16,596	42
ff4j[1]	346.92s	119.29s	34.39%	220.44s	63.54%	15,485	43
jetbrick[20]	39.01s	25.06s	64.24%	13.55s	34.73%	70	11
styx[16]	381.87s	129.91s	34.02%	234.73s	61.47%	6,862	36
geowave[13]	72.68s	35.50s	48.84%	33.99s	46.77%	12,768	18
Avg.	345.06s	115.32s	33.42%	212.61s	61.62%	11,515	36

TPL_CE: *The time of analyzing TPLs to generate call edges.*
RA: *The query time of reachability analysis.*
R_1: *The proportion of the time* (TPL_CE/sum).
R_2: *The proportion of the time* (RA/sum).
Q.: *The number of queries for the detection.*
D.: *The number of dependency trees for the project.*
The time required for loading the runtime environment is not listed.

directly tell which nodes are reachable from the TPL A. Therefore, the adaptation of existing indexing approaches is not practical enough to be deployed in IDEs.

2.4 Key Idea

To effectively reduce the time required for graph reachability analysis, we design a compositional reachability analysis method for call graphs in the presence of TPLs,'' which includes the offline and online analyses. In the offline analysis, for each TPL, we perform call graph analysis to collect all call edges and generate the summaries to store in the disk. When answering the call graph reachability query online, we retrieve the summaries of necessary TPLs on demand from the disk, so as to reduce the computation overhead.

In the offline analysis, we propose to leverage TC which essentially encodes the reachability information among all nodes as the summaries. Unlike existing graph indexing approaches, TC supports answering the question of which nodes are reachable from a given node in the call graph, thus aiding in establishing connections among different TPLs. For instance, in the call graph of the example code shown in Figure 2, for the TPL A, our initial input consists of every edge within the library, including the edges ③, ④, ⑤, ⑥ and ⑦. We define the graph formed by combining these edges as the internal call graph, with a detailed definition in Section 3.2. After computing the TC, we determine that `foo_a` can reach both `bar_b`, `cor_a` and `foo_c`, corresponding to the edges ⑨, ⑩ and ⑪. The edges ⑨ (connecting TPLs A and B) and ⑪ (connecting TPLs A and C) are identified as the connecting edges between the TPLs. Through these edges, we can establish connections with other libraries. The

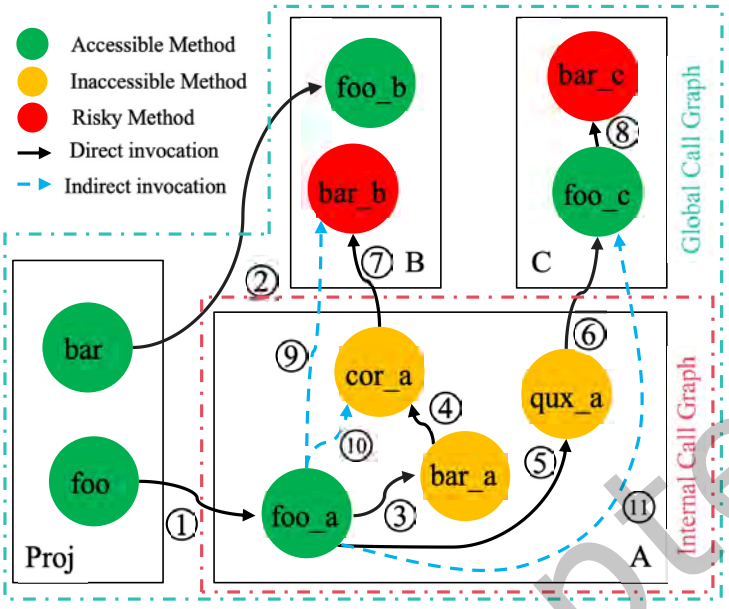

Fig. 2. Two kinds of call graphs of the motivating code example in Figure 1

global call graph combines all call edges from the entire project, including TPLs. A detailed definition can be found in Section 3.2.

When answering the reachability query within the project utilizing TPLs, we retrieve the summaries of the TPLs on demand, based on which the reachability relation can be determined. Instead of indiscriminately loading all TPLs, we identify the TPLs relevant to the reachability query based on the dependency relations among TPLs, selectively loading summaries of TPLs on the dependency paths for composition. Taking Figure 2 as an example, to determine if the project code can reach the risky method in TPL *C*, we selectively load only the summaries of TPLs *A* and *C* which are directly involved in the reachability path from Host to `bar_c`. This design significantly reduces the time overhead.

In the following sections, we first introduce the preliminaries and formulate the problem in Section 3. Then we demonstrate the technical details of our compositional call graph reachability analysis in Section 4. After illustrating the applications and implementations of our approach in Section 5, we present the evaluation results in Section 6, which provide sufficient empirical evidence to demonstrate the effectiveness of our approach.

3 PROBLEM FORMULATION

This section first presents the program syntax (Section 3.1) and then formulate call graph models for applications using TPLs (Section 3.2). At the end of the section, we state the call graph reachability problem (Section 3.3).

3.1 Program Syntax.

We formalize the project syntax in Figure 3. We define a project *P* as a set of source code files *F* and a collection of TPLs used in the development of the project. In this paper, we also consider a TPL as a project that contains multiple source code files and their dependent projects. Within each source code file *F*, at least one method *M* is defined. A method *M* is characterized by a modifier, a fully qualified name, and a sequence of statements.

$$\texttt{Project}\ P := \{F^+ \mid P^*\}$$
$$\texttt{Source File}\ F := M^+$$
$$\texttt{Method}\ M := (\text{modifier},\ \text{name},\ Stmt^+)$$
$$\texttt{Statement}\ Stmt := c \mid \ldots$$
$$\texttt{Function Call}\ C := \textbf{invoke}(m)$$

Fig. 3. The project syntax.

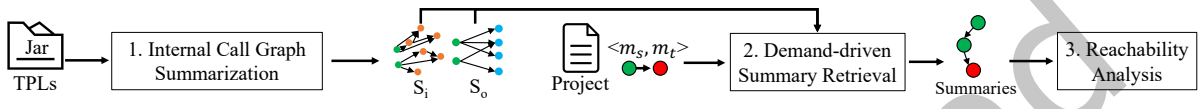

Fig. 4. The workflow of ReachCheck

Particularly, a modifier indicates the accessibility that whether a method is `public`, `protected`, or `private`. When analyzing the statement in a method, we only focus on the function call statements. It is worth noting that the aim of our work is to target the reachability analysis upon the call graph. Our work primarily assumes the caller-callee relation as a prerequisite, taking for granted that all possible callee functions for each function call statement are determined, rather than delving into additional complex analyses like class-hierarchy and pointer analysis.

3.2 Call Graph Models.

To facilitate the formulation of call graph reachability problem, we introduce two important call graph models as follows, which formalize different forms of caller-callee relations in a project using TPLs.

Definition 3.1. (Internal Call Graph) Given a project P, its internal call graph G is a pair $(M,\ E)$, where M and E are the sets of the methods and call edges, respectively. Specifically, a method $m \in M^i$ can be defined in P or $m \in M^o$, offered by a TPL of P. A call edge $e \in E$ has the following two forms:

- $e := (m_s, m_t) \in E^i \subseteq E$ is an *inner call edge* if $m_s \in M^i$ and $m_t \in M^i$.
- $e := (m_s, m_t) \in E^o \subseteq E$ is an *outer call edge* if $m_s \in M^i$ and $m_t \in M^o$.

Example 3.2. An example of the internal call graph in Figure 2 is the call graph for the TPL A in the code example in Figure 1. The TPL A has five call edges: ③, ④, ⑤, ⑥ and ⑦. Specifically, the edges ③, ④ and ⑤ are inner call edges, while the edges ⑥ and ⑦ are outer call edges.

The internal call graph follows the standard formulation of caller-callee relation. In the library-aware program analysis, we also need to analyze the program constructs in the TPLs. Hence, we introduce the second call graph model to offer a global perspective for analyzing call graph reachability upon the projects using TPLs.

Definition 3.3. (Global Call Graph) Consider a project P using a set of TPLs $\mathcal{L} = \{L_i \mid 1 \le i \le n\}$ $(n \ge 1)$. Assume the internal call graphs of P and each TPL L_i are $G_P = (M_P, E_P)$ and $G_{L_i} = (M_{L_i}, E_{L_i})$, respectively. Then the global call graph of P is $\widetilde{G} := (\widetilde{M}, \widetilde{E})$ defined as follows:

$$\widetilde{M} = M_P \bigcup \Big(\bigcup_{1 \le i \le n} M_{L_i} \Big), \quad \widetilde{E} = E_P \bigcup \Big(\bigcup_{1 \le i \le n} E_{L_i} \Big)$$

Algorithm 1: Reachability Check via DFS

Require: A global call graph $\widetilde{G} = (\widetilde{M}, \widetilde{E})$
Require: A source method m_s and a target method m_t
Ensure: Whether there exists a path from m_s to m_t in $\widetilde{G}$

```
1:  function ReachabilityCheck(G̃, m_s, m_t)
2:      visited ← [m ↦ false | m ∈ M̃]
3:      return DFS(m_s, m_t, visited)

4:  function DFS(m, m_t, visited)
5:      if m = m_t then return true
6:      visited[m] ← true
7:      for each n ∈ neighbors(m) do
8:          if not visited[n] and DFS(n, m_t, visited)
9:              then return true
10:     return false
```

Basically, the global call graph is stitched from the internal call graphs of the project and its TPLs. In particular, the outer call edges in each internal call graph depict caller-callee relations across the project and TPLs. Notably, all the methods and caller-callee pairs induced by the function calls in the project and TPLs are completely encoded in the global call graph. Hence, it can serve as the ingredient for examining call graph reachability in a library-aware manner.

Example 3.4. Figure 2 shows four internal call graphs of the project `Proj` and its three TPLs, each of which is located in a rectangle zone. There are four outer call edges, including ①, ②, ⑥, and ⑦. Ultimately, all the methods, inner call edges, and outer call edges form the global call graph $\widetilde{G}$ of the project `Proj` using the three TPLs.

3.3 Problem Statement.

As formulated in Definitions 3.1 and 3.3, the two call graph models are important intermediate representations of a project using TPLs. By conducting specific downstream client analyses upon them, we can understand how a method in the current project invokes a method in a TPL, which has significant impact in improving software security and quality. For example, if there is a path from a method in the current project to a risky method in a TPL, we can detect a potential vulnerability caused by the usage of insecure TPLs. For example, as reported in Issue#309 [1] of GitHub project `ff4j/ff4j` [11], the developer does not know that they are indirectly using a risky method in a TPL, while a security expert finds this potential call path through call graph analysis. Hence, it is a fundamental problem to reason whether one method can invoke another method directly or transitively. We formulate it as call graph reachability problem as follows.

Definition 3.5. (Call Graph Reachability Problem) Given a project P using a set of TPLs $\mathcal{L} = \{L_i \mid 1 \leq i \leq n\}$ ($n \geq 1$), and a query pair (m_s, m_t) where the source method m_s from P, and the target method m_t from L, compute the global call graph $\widetilde{G}$ and determine whether there exists a path p from m_s to m_t in $\widetilde{G}$.

To address the reachability query problem on call graphs, a direct approach involves using online traversal algorithms, such as DFS or BFS algorithm, upon a constructed global call graph. Algorithm 1 formalizes the call graph reachability analysis via DFS. For each query pair of m_s and m_t, it continuously traverses each edge of the global call graph until it finds a path from m_s to m_t. However, as the size of the graph increases, there can be

substantial time overhead in the presence of many TPLs. While several caching strategies can help reduce the overhead, maintaining a global call graph for the entire project is fragile, any change to the version of a TPL would invalidate the entire cache and require full recomputation. A natural alternative is to cache the call graph of each TPL separately. However, as shown in our evaluation, this approach still incurs high overhead in IDE settings and is not suitable for interactive, incremental analysis. In this work, we aim to develop an approach that accelerates call graph reachability analysis while being resilient to changes in both TPLs and project code. This can directly improve the efficiency of many downstream clients, including dependency conflict detection and CVE risk detection. In the next section, we will demonstrate the technical details of our approach ReachCheck, a novel compositional library-aware call graph reachability analysis that addresses the reachability queries of method pairs in the IDE.

4 APPROACH: COMPOSITIONAL CALL GRAPH REACHABILITY ANALYSIS

This section presents the technical details of our approach ReachCheck, of which the workflow is demonstrated in Figure 4. Specifically, ReachCheck consists of three steps. First, we generate internal call graph summaries for each TPL (Section 4.1). Then, we retrieve the call graph summaries on-demand based on the project and query pairs (Section 4.2). Finally, we introduces an efficient reachability analysis based on the call graph summaries (Section 4.3). In what follows, we demonstrate each step in detail.

4.1 Internal Call Graph Summarization

We aim to design a new call graph summary structure that addresses existing limitations, specifically the inefficiency of current call graph reachability analyses and the inability of indexing approaches to adapt to changes in the call graph. The overarching idea is pre-computing their reachability relation as the summaries. When performing reachability queries, we retrieve the pre-computed summaries on-demand for each TPL, which allow for efficient reachability analysis between any two methods subsequently. We introduce the concept of *internal call graph summary*, which formulates the reachability relations over an internal call graph.

Definition 4.1. (Internal Call Graph Summary) Assume the internal call graph of a project or a TPL is $G = (M^i \cup M^o, E)$, where M^i and M^o are defined in Definition 3.1. Its internal call graph summary S is a pair (S_i, S_o), where

- $S_i \subseteq M^i \times M^i$ is a set of *inner summaries*. $(m_s, m_t) \in S_i$ indicates each method m_s in the current project or TPL can directly or transitively invoke another method m_t.
- $S_o \subseteq M^i \times M^o$ is a set of *outer summaries*. $(m_s, m_t) \in S_o$ indicates each method m_s in the current project or TPL can directly or transitively invoke the method m_t defined in other TPLs.

Example 4.2. In Figure 2, the method `foo_a` includes two direct call edges (③ and ⑤), and we can compute the indirect call edge ⑩ introduced transitively via ③ and ④. Similarly, we can compute the two edges ⑨ and ⑪. These reachable methods are recorded in `foo_a` to create a call graph summary. With this summary, we can quickly ascertain which locations any given method in G can reach. For example, method `foo_a` can directly reach `cor_a`. Among them, ③, ⑤, and ⑩ are part of S_i, while ⑨ and ⑪ are part of S_o.

Intuitively, the internal call graph summary depicts the reachability relation upon the internal call graph from two aspects. Specifically, an inner summary indicates the reachability relation between two methods in the project or a TPL, while an outer summary concentrates on whether a method in the project or a TPL invokes the methods in other TPLs. In the current definition, such two kinds of reachability relations are decomposed, which facilitates an efficient pre-computation of the internal call graph summary, which is formulated in Algorithm 2.

Technically, Algorithm 2 computes the internal call graph summary by applying the fast matrix multiplication technique. First, we extract inner call edges E^i and outer call edges E^o from the internal call graph (line 1). Then,

Algorithm 2: Internal Call Graph Summarization

Require: Internal call graph G
Ensure: Internal call graph summary S_i and S_o
1: **function** GenerateSummary(G)
2: $(E^i, E^o) \leftarrow G$
3: $\mathbb{M}_i \leftarrow$ ConstructMatrix(E^i)
4: $\mathbb{M}_j \leftarrow$ MatrixMultiplication($\mathbb{M}_i, \mathbb{M}_i$)
5: **while** $\mathbb{M}_i \neq \mathbb{M}_j$ **do**
6: $\mathbb{M}_i \leftarrow \mathbb{M}_j$
7: $\mathbb{M}_j \leftarrow$ MatrixMultiplication($\mathbb{M}_i, \mathbb{M}_i$)
8: $S_i \leftarrow$ RemoveInaccessible($\mathbb{M}_i$)
9: $S_o \leftarrow$ CalcOuterSummary(S_i, E^o)
10: **return** S_i, S_o

we construct the computation matrix $\mathbb{M}_i$ according to the inner call edges (line 2). Notably, we only concentrate on the inner call edges in constructing $\mathbb{M}_i$ because they sufficiently capture essential connectivity within the internal call graph, thereby minimizing redundant computations. The outer call edges E^o that connect to methods in another TPL, such outer methods are terminal, as they can not invoke any methods in the current project and TPL. As a result, when computing the TC, we can focus on inner call edges and combine them with outer call edges to compute the summary for external connections. This approach significantly reduces the computational overhead involved in deriving the inner summary (lines 4-7). Here, the function MatrixMultiplication represents the multiplication of the matrix itself, which is achieved by the fast matrix multiplication. Next, we can eliminate the rows $\mathbb{M}_i$ corresponding to inaccessible methods from the TC matrix to derive the inner summary S_i (line 8). Overall, by computing the TC solely over inner call edges, we significantly reduce the size of the TC matrix, thereby improving the efficiency of the computation. **We call this optimization strategy as *call edge refinement.*** The impact of this strategy on our algorithm is further discussed in Section 7.1. Notably, preserving the transitivity of inaccessible methods is unnecessary since these methods will not be invoked by other methods outside the current TPL. This optimization reduces the storage space required for the summary. Building upon the inner summary, we can obtain the outer summary S_o by concatenating them with the outer call edges in E_o (line 9). Compared to directly computing the reachability relationships across all methods $M^i \cup M^o$, this approach enables fast matrix multiplication on smaller matrices, accelerating the analysis and minimizing space overhead.

Example 4.3. In (a) of Figure 5, shown the call graph matrix of the code example TPL A as shown in Figure 2. The green functions indicate methods accessible outside the package, such as functions with a `public` access modifier. In contrast, the red functions represent methods inaccessible outside the package, such as functions with a `private` access modifier. We compute the internal call graph summary based on this call graph matrix. The black *1* indicates direct call edges and the empty positions correspond to the absence of a call relationship, represented as *0* in the matrix. First, we construct an initial matrix according to all inner call edges in A, for example, in (a) of Figure 5, the (4×4) matrix from row `foo_a` to `qux_a` and column `foo_a` to `qux_a`. We then use matrix multiplication to compute the final TC matrix. In (b) of Figure 5, the blue *1* represent call edges obtained through TC computation. To save computational space, we remove three rows of data from `bar_a` to `qux_a` because they are inaccessible from external methods in other TPLs, retaining only the (1×3) denoted as S_i, shown in (c) of Figure 5. Subsequently, by combining the outer call edges, shown in (c) with S_i, we compute the `foo_a` to `bar_b` and `foo_c`, the blue *1* in (c). Finally, we remove redundant summary edges, similar to obtain S_i, to calculate the final S_o, shown in (d) of Figure 5.

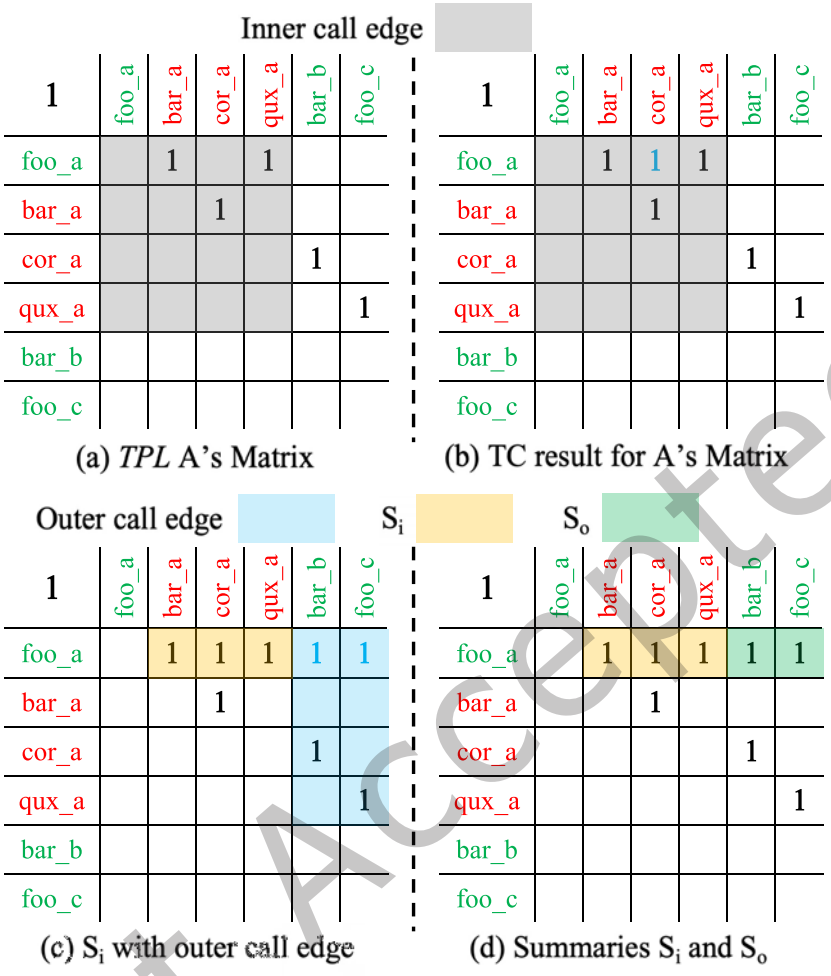

Fig. 5. An example of computing the internal call graph summary for TPL *A* (Green: `public` methods, Red: `private` methods).

The decomposition of call graph summaries into S_i and S_o effectively mitigates the risk of query explosion. For each TPL, ReachCheck only needs to answer two kinds of reachability queries. First, it identifies which internal methods within the TPL are reachable from a given entry method by S_i. Second, it determines which external methods are defined in other libraries—can be reached from the internal methods by S_o. By separating internal and external reachability into distinct summaries, ReachCheck avoids the need to enumerate all possible method pairs, significantly reducing the computational cost. During online reachability queries, we leverage the dependency path (see Section 4.2) to guide the composition of summaries. When resolving a query to a target method in a transitive dependency, we intersect the reachable outer methods from the current TPL (S_o) with the entry points of the next TPL along the dependency path. This process significantly reduces the number of required query pairs by pruning irrelevant call chains early and avoiding redundant exploration of methods not involved in the reachability path.

Algorithm 3: Demand-driven Summary Retrieval

Require: Project $P = \{F+, P*\}$, query target method m_t
Ensure: Inner call graph summary $\widetilde{S_i}$ for the `last`(D_p)
Ensure: Outer call graph summaries set $\widetilde{S_o}$
1: **function** GetSummary(P, m_t)
2: $Tree \leftarrow$ BuildDependencyTree(P)
3: $D_p \leftarrow$ GetDependencyPath$(Tree, m_t)$
4: **for each** $L \in D_p$ **do**
5: **if** L == `last`(D_p) **then**
6: $\widetilde{S_i} \leftarrow$ GetInnerSummary(L)
7: **else**
8: $\widetilde{S_o} \leftarrow \widetilde{S_o} \cup$ GetOuterSummary(L)
9: **end if**
10: **end for**
11: **return** $\widetilde{S_i}$ and $\widetilde{S_o}$

It is worth remarking that the internal call graph summary computation is a one-time effort for a TPL and can be achieved offline. Benefiting from the decomposition of two kinds of summaries and other optimization techniques, such as the call edge refinement and fast matrix multiplication, we significantly reduce the complexity of the TC computation, thus inducing low time and space overheads. Based on the pre-computed internal call graph summaries, we can avoid redundantly reasoning the call graph reachability when analyzing different projects using the same TPLs.

4.2 Demand-driven Summary Retrieval

As demonstrated in Section 4.1, the internal call graph summaries of the project and its TPLs support discovering reachable paths within the global call graph. While the internal call graph summaries can be computed with one-time effort offline, loading all internal call graph summaries for each reachability query can result in significant overhead due to the large number of TPLs. To mitigate memory consumption, we propose the concept of a dependency path, which facilitates a demand-driven retrieval of internal call graph summaries.

Definition 4.4. (Dependency Path) Given a project P and a target method m_t within a TPL, the dependency path D_p is defined as an ordered list of packages that represents the traversed path from project P to the TPL that contains the target method m_t. Specifically, a TPL L in the dependency path D_p has the following two forms:

- L_t is an intermediate TPL, corresponding to an intermediate node in the dependency path.
- L_e is a terminal TPL that contains the target method m_t.

To convenience the formulation, we introduce `last`(D_p) to indicate the terminal TPL in the dependency path.

Example 4.5. As shown in Figure 2, assume that m_t is a risky target method `bar_c` located in a TPL C. Therefore, the dependency path D_p includes all the TPLs from `Proj` to C, and can be represented as an ordered list $[A, C]$, where A is the intermediate TPL and C is the terminal TPL.

Algorithm 3 provides a demand-driven method for retrieving call graph summaries based on a specified project P and query target method m_t. Initially, the algorithm constructs a dependency tree for P (line 3) and uses it to determine the dependency path D_p (line 4), which links P to the TPL containing m_t. The dependency path D_p is derived through a backward traversal, starting from the TPL containing m_t and moving up through its

Algorithm 4: Reachability Analysis

Require: Query pair (m_s, m_t), dependency tree path D_p
Require: Call graph summaries $\widetilde{S_i}$ and $\widetilde{S_o}$
Ensure: Whether m_t is reachable from m_s
1: **function** ReachAnalysis($m_s, m_t, D_p, \widetilde{S_i}, \widetilde{S_o}$)
2: $M^S \leftarrow \{m_s\}$
3: **for each** $L \in D_p$ **do**
4: **if** $L \neq$ `last`(D_p) **then**
5: $M^S \leftarrow$ CalcNextReach($M^S, \widetilde{S_o}, L$)
6: **else**
7: **return** CanReach($M^S, m_t, \widetilde{S_i}$)
8: **end if**
9: **end for**
10: **return** *False*

parent nodes until reaching P. For each L in D_p, the algorithm checks if it is the last node in the dependency path, indicating that it contains m_t. If it is, the algorithm retrieves the inner call graph summary $\widetilde{S_i}$ for L (lines 6-7). Otherwise, it fetches the outer call graph summary $\widetilde{S_o}$, which records reachable external methods (lines 7-8). Selecting specific summaries based on the position of TPL in the dependency path D_p can effectively reduce the summary size needed for reachability queries, thereby improving query efficiency. **We call this optimization strategy *on-demand sub-summary retrieval.*** The impact of this strategy on our algorithm is further discussed in Section 7.2. The final output includes $\widetilde{S_i}$ and $\widetilde{S_o}$ for TPLs on the dependency path (line 10). This selective retrieval method reduces memory usage by loading only the necessary summaries based on D_p, thus optimizing reachability queries.

Example 4.6. For the query pair (`foo`, `bar_c`) shown in Figure 2, the TPL dependency path is identified as $[A, C]$, following the backward traversal as outlined in Definition 4.4. In this path, A is classified as the transit node L_t, while C is the terminal node L_e since it contains the target method `bar_c`. To resolve this query, we selectively retrieve specific summary matrices for each TPL along the dependency path. For node A, we use the outer summary matrix $\widetilde{S_o}$ to identify methods in other TPLs that are reachable from methods within A. Specifically, edges ⑨ and ⑪ are included in $\widetilde{S_o}$, indicating that `foo_a` in A can reach methods outside of A via these edges. The remaining edges within A are excluded from $\widetilde{S_o}$, as they are irrelevant to this particular reachability query. For C, we utilize the inner summary matrix $\widetilde{S_i}$, which encapsulates the internal reachability among methods within C. This matrix allows us to verify reachability from `foo_c` to `bar_c` specifically, represented by edge ⑧ in $\widetilde{S_i}$. By focusing solely on these specific summary matrices $\widetilde{S_o}$ for A and $\widetilde{S_i}$ for C, we optimize the query process by minimizing the amount of data retrieved and processed.

In this step, we optimize the retrieval of call graph summaries by constructing a dependency path and retrieving summaries based on the position of TPL nodes in this path. This selective retrieval reduces memory usage and computational overhead, optimizing the reachability analysis by focusing only on the necessary summaries.

4.3 Reachability Analysis

As demonstrated in Section 4.2, we obtain the project's D_p based on the target TPL with m_t and then demand-driven retrieve relevant summary along this path. The summaries primarily include the $\widetilde{S_i}$ of the last dependency

node in the path and the $\widetilde{S_o}$ of all other nodes along the path. We can quickly and efficiently determine API reachability with these summaries while minimizing the amount of summaries used.

To determine if a target method m_t is reachable from a source method m_s, we employ Algorithm 4. This approach leverages dependency path D_p and precomputed call graph summaries $\widetilde{S_i}$ and $\widetilde{S_o}$ to efficiently assess reachability. The process begins by initializing $M^S = \{m_s\}$ (line 2), where M^S represents the set of methods that are reachable from m_s as the algorithm progresses. This set M^S is updated iteratively as each L in D_p is processed, expanding to include methods that can be accessed from m_s by traversing through intermediate methods (lines 3-9). Specifically, if the L is not the last one in the dependency path, we then utilize the `CalcNextReach` function to compute the set of methods reachable from M^S using $\widetilde{S_o}$, updating M^S to recollect this new set of reachable methods (lines 4-5). Specifically, in the `CalcNextReach` function, we select the outer call graph summary corresponding to the L in the set of outer call graph summaries $\widetilde{S_o}$ for computation. Otherwise, the algorithm utilizes `CanReach` to verify if m_t lies within the reach of M^S in $\widetilde{S_i}$ (lines 6-7). This selective updating process optimizes reachability analysis by limiting computations to necessary summaries.

Example 4.7. In Figure 2, consider `bar_c` as a risky target method m_t. Then, consider `foo` as a source method m_s in `Proj`. The D_p is $[A, C]$. The first TPL in the D_p is A, which is L_t. Thus, we use its $\widetilde{S_o}$ in (d) in Figure 5. Based on the analysis of the `Proj`, `foo` can reach to `foo_a`, while *bar* does not call any methods in A. We combine M^S with $\widetilde{S_o}$, resulting in an updated M^S that includes the methods `bar_b` and `foo_c` to be propagated to the next node. The next TPL, C, is L_e, so we retrieve its $\widetilde{S_i}$. Combining M^S with $\widetilde{S_i}$, we ultimately reach the risky method `bar_c`. Thus, we conclude that it is possible to call the risky method `bar_c` from `foo` in `Proj`.

In summary, the reachability analysis effectively leverages precomputed call graph summaries to expedite node reachability determination. Our approach minimizes computational overhead and enhances query efficiency by utilizing summaries along the dependency path, enabling faster, more efficient reachability queries.

5 APPLICATIONS AND IMPLEMENTATION

This section presents two applications of reachability detection and provides the implementation details.

5.1 Applications of Call Graph Reachability Analysis

To demonstrate the capability of ReachCheck, we implemented two core applications: dependency conflict detection and CVE risk detection (See Section 2.2). Both applications are based on identifying risk methods. Once identified, ReachCheck can perform a reachability analysis to assess the reachability of these methods. For dependency conflict detection, we identify risks by comparing the lists of accessible methods between two library versions, highlighting any accessible method differences. Variations in the accessible method lists of the TPL between the two versions are treated as risk methods that require attention. For CVE risk detection, we establish a CVE vulnerability database focusing on Java TPLs. The database mainly includes CVE risks and the range of versions of libraries affected by specific risk methods. We obtain CVE risk data from public vulnerability database Snyk [3], a widely used security database that tracks and identifies vulnerabilities in open-source dependencies. We then filter risk-related methods based on GitHub commits associated with risk fixes, flagging them as target risk methods for detection.

5.2 Implementation

We conduct the internal call graph summarization analysis offline on the server. We apply a call graph analysis algorithm (e.g., Class Hierarchy Analysis (CHA) or Variable Type Analysis (VTA)) to analyze each TPL's bytecode, extracting method signatures and call edges from public methods which are treated as entry points of call graph analysis. Subsequently, we generate the internal call graph summary of TPL using cuBool [44, 48] via matrix

multiplication, enhanced by GPU parallel computing for efficiency. To improve preprocessing efficiency, we utilize GPU parallelism during this one-time offline summarization phase. Note that GPU acceleration is only applied at this stage. The actual reachability queries can be executed on the CPU, and the performance improvements primarily stem from the summary-based design rather than hardware optimization. The summary is then stored in a central repository similar to the Maven repository to facilitate user access.

To support incremental updates and avoid recomputing summaries when a TPL is updated, ReachCheck incorporates a mechanism for efficiently handling newly introduced subclasses. Updating a TPL version or adding a new TPL may introduce new subclasses, which in turn can result in additional virtual calls to existing call sites. To address this without requiring full recomputation, ReachCheck records call edges from downstream libraries to interface types and abstract classes during summary construction. When new subclasses are introduced, ReachCheck consults the recorded type usage information to identify affected call sites and triggers an on-demand reachability query starting from the newly introduced virtual call edge. In this process, we conservatively use the CHA algorithm for call graph analysis to add new call edges during query time. This algorithm is independent of the call graph algorithm used during the initial summary construction. This mechanism enables ReachCheck to preserve the efficiency of summary reuse.

To support the analysis of callback, ReachCheck adopts an incremental query strategy. Specifically, when a TPL invokes a callback method which is defined in the client project, we treat this as a new reachability query initiated from the client project method to the target methods. Since each individual query is executed very efficiently, these small incremental queries introduce negligible time overhead.

Our approach enables simultaneous querying of the reachability of all target methods within a given TPL. By leveraging a matrix model, our approach supports inputting query pairs as sets, where M^S represents all methods m_s in the project `Proj`, and M^T denotes the target methods m_t for querying within the TPL. This structure allows our approach to efficiently determine reachability between the sets M^S and M^T by utilizing call graph summaries, significantly reducing query time. For example, in the case shown in Figure 2, we can group all methods in `Proj`, such as foo and bar, into the set M^S, while M^T would consist of foo_c and bar_c as target methods. By executing the reachability query algorithm once, we can quickly determine which methods in TPL C are reachable from `Proj`. This approach avoids generating all possible query pairs ($foo \rightarrow bar\_c$, $bar \rightarrow bar\_c$, et. al), thereby reducing redundant computations. This allows us to assess the reachability of risky methods more efficiently.

6 EVALUATION

In this section, we evaluate the efficacy of ReachCheck by answering the following research questions.

- **RQ1:** How efficiently does ReachCheck perform in querying call graph reachability?
- **RQ2:** Can ReachCheck outperform state-of-the-art other reachability approaches in terms of query time efficiency?
- **RQ3:** Can ReachCheck assist in dependency conflict detection and CVE risk detection to meet the IDEs' requirements?

6.1 Experimental Setup

Dataset. To address RQ1 and RQ2, we constructed a dataset of API pairs for call graph reachability queries with the following steps. First, we randomly selected 100 widely-used and well-known Maven projects from GitHub, where their source code and their dependent TPLs are all available. The selected projects have at least 50 stars, forks, update frequently. For each project, we leveraged Maven to download all TPLs it depends on. Our evaluation setting is designed to simulate the interactive development scenario in IDEs, where a developer introduces a new call (e.g., writes a line of code), which immediately triggers a reachability query. In such cases, queries are independent and arrive one at a time. Then, we randomly sampled 2.5 million reachability queries

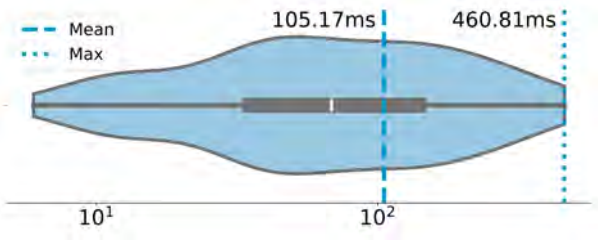

Fig. 6. The execution time of ReachCheck.

(m_s, m_t) for each project, where each $m_s \in M^S$ belongs to the project and each $m_t \in M^T$ is part of the project's TPLs. For simplicity, we refer to this dataset as *ReachBench*[24]. The maximum number of dependencies of these projects is up to 224, and the average is 71, including direct and indirect dependencies, ensuring a diverse and representative sample.

Baselines. We compare ReachCheck with three baselines. It should be noted that the source code for these approaches is open-source, and we use their configurations that are used as the best in the paper for comparison.

- **Online traversal (OT) approach:** It is an approach that builds the call graph of the project and the required TPLs and leverages BFS/DFS traversal algorithm to answer the call graph reachability query, which has been widely used in existing approaches, e.g., Decca [67] and Riddle [68].
- **Function summary (FS) approach:** It is an approach that pre-analyzes and caches call graphs of the required TPLs offline and and leverages BFS/DFS traversal algorithm to answer the call graph reachability query [43]. Due to the pre-analysis, it can save the analysis time of online building call graph compared with the OT approach.
- **Graph indexing approaches:** They are the approaches that build graph index for the required TPLs and leverage a graph index to answer the call graph reachability queries. There are three categories of graph indexing approaches, including tree-cover [26], 2-hop labeling [32], and approximate TC [71, 80]. We compared ReachCheck with the state-of-the-art techniques of these three categories, namely Ferrari [56], BL [78], and BFL [61].

It should be noted that, in our experiment, we verify the answers to call graph reachability queries among all the aforementioned approaches are consistent, so as to ensure the correctness of the implementations for ReachCheck and all the baselines. For any inconsistency, we identify the implementation issues and fix them until they produce consistent results.

Environment. Our experiments were conducted in two separate environments to reflect realistic usage settings and ensure fair evaluation:

- **Summary Generation.** The one-time offline summary generation phase, which includes matrix-based TC computation, was performed on an NVIDIA RTX A6000 GPU (48GB VRAM) running Ubuntu 22.04.1 LTS. The GPU was used exclusively for accelerating matrix computations during summary construction, and was not involved in query execution.
- **Reachability Query.** To reflect realistic IDE scenarios, all reachability queries and baseline comparisons were conducted on a local workstation equipped with an 8-core "11th Gen Intel(R) Core(TM) i7-11700@2.50GHz" processor and 64GB of RAM. All experiments were executed on a single thread without any parallelism.

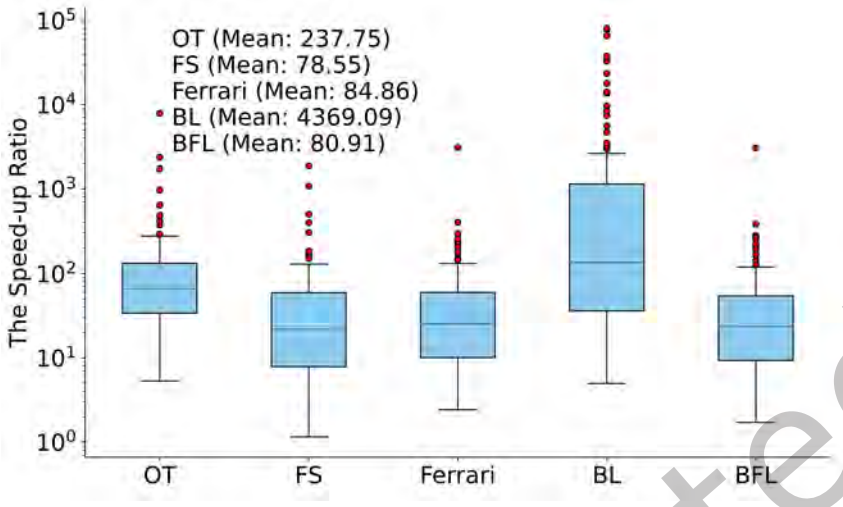

Fig. 7. The speedup compare to other approaches.

6.2 Efficiency

To evaluate the efficiency of ReachCheck, we measure its execution time of answering the queries in the dataset *ReachBench*. Figure 6 shows the execution time by ReachCheck for answering all reachability queries across 100 Maven projects. Across all projects in the dataset, ReachCheck required an average of 105.17 ms to complete the reachability queries per project. For example, in the project *PGM* that requires the longest query time, ReachCheck completes all 2.5 million queries in just 460.81 milliseconds. This performance indicates that ReachCheck can process over 5.43 million reachability queries per second, demonstrating its ability to meet the performance requirements of various analysis tasks in an IDE environment. Consider the two applications described in this work. First, the dependency conflict detection shown in Table 2 involves no more than 62,098 reachability queries. Second, for CVE risk detection, even with 1,000 call sites as sources and 1,000 known vulnerable methods as targets, the analysis would require only 1 million queries. Given these numbers, ReachCheck is fully capable of handling both tasks efficiently within an IDE's performance constraints.

Answer to RQ1: ReachCheck efficiently answers 2.5 million call graph reachability queries per project, with an average time cost of 105.17 ms, in line with the performance requirements expected within IDEs.

6.3 Comparison with baselines

We compared ReachCheck with the three categories of the representative baselines as discussed in Section 6.1, namely the online traversal approach, function summary approach, and graph indexing approaches. Figure 7 illustrates the speedup of our approach over the other baselines.

Compared to the online traversal approach, our approach ReachCheck achieved a speedup of 237.75× on average. Specifically, our approach achieves far more significant speedups in large-scale projects than smaller ones. For example, in large-scale projects like *IoT-Technical* and *line-login*, with call edges ranging from 2,513,225 to 5,172,866, ReachCheck achieves speedups of 289.03× and 201.77×, respectively. In contrast, for smaller projects such as *ymate-platform-v2* and *perwendel-spark*, which have call edges ranging from 37,309 to 43,833, the speedups

are 93.87× and 110.82×. The function summary approach is essentially more efficient than the online traversal approach, since it reduce the time cost of online computing call graphs of TPLs. Even so, our approach is still more efficient than the function summary approaches, with the average speedup of 78.55×. The result is not surprising, because ReachCheck not only saves the time of computing call graph but also leverages TC summaries to speed up the reachability queries.

Compared to the three representative graph indexing approaches, i.e., Ferrari, BL and BFL, respectively, ReachCheck achieves average speedups of 84.86×, 4, 369.09× and 80.91× in *ReachBench* [24]. The primary reason for this substantial improvement is that ReachCheck leverages pre-built call graph summaries for each TPL, enabling efficient demand-driven combination and support for fast reachability queries. Compared to ReachCheck, existing graph indexing approaches incur substantial overhead due to the need to construct a complete call graph index by merging the call graphs of all TPLs. In our evaluation, the baseline methods report the total analysis time, which includes two parts. The first part is the time required to merge all TPL call graphs into a complete call graph. The second part is the time spent performing reachability queries. Even when comparing only the reachability query phase, ReachCheck still achieves substantial performance gains. Specifically, it outperforms the baseline indexing methods with speedups of 8.41× over Ferrari, 4, 292.64× over BL, and 4.46× over BFL, respectively. The BL graph indexing approach performs the worst due to its design, which assumes to work efficiently on sparse graphs with an expected average degree of less than 2 [78]. However, such assumption does not hold in the scenario of call graphs. For example, in the projects of *ReachBench*, the average degree is 5.13, which breaks the assumption of of the BL graph indexing approach and thus undermines its efficiency.

Answer to RQ2: ReachCheck significantly outperforms the state-of-the-art reachability approaches in terms of query time efficiency, achieving speedups of up to 78.55× compared to the fastest baseline and demonstrating its strong potential for accelerating detection tools to meet the IDE requirement.

6.4 Usefulness

We evaluated the usefulness of ReachCheck using two downstream tasks: dependency conflict detection and CVE risk detection. To ease the explanation, we named the tools implemented based on ReachCheck for dependency conflict detection and CVE risk detection as *DepConDetect* and *CVERiskDetect*, respectively.

Dependency conflict detection: We reproduced the Maven projects in the issues reports from the paper [67, 68]. To better understand the efficiency of *DepConDetect*, we used Decca, which is essentially based on the online traversal approach, for the comparison. The evaluation results indicate that *DepConDetect* can reduce the average detection time for each reported issue from 327.93 seconds to 0.61 seconds when detecting the same issue. To facilitate the reproduction, we have published the data and the evaluation results online [6]. This demonstrates that ReachCheck can effectively accelerate dependency conflict detection tasks, thereby meeting the requirements of the IDE environment.

CVE risk detection: We utilized *CVERiskDetect* to detect Maven projects in *ReachBench* with the following steps. First, we retrieved the methods that are associated with CVEs, namely CVE methods, from our constructed CVE database in Section 5.1. For each Maven project, we checked whether it uses any risky TPL which contains the CVE by matching the TPL name and version with the records in the CVE database. If matching, we extracted the CVE methods contained in the risky TPL as M^T. Finally, we leveraged *CVERiskDetect* to determine whether M^T are reachable from any method in the project. Eventually, we submitted 43 bug reports as shown in Table 3[5]. All of the bug reports have been confirmed, and 40 of them have been fixed.

To better understand the efficiency of *CVERiskDetect*, we implemented a baseline CVE detection tool based on online traversal approach, namely, *CVERiskDetect_OT*. The results demonstrated that, comparing with *CVERiskDetect_OT*, *CVERiskDetect* significantly reduced the detection time from 181.66 seconds to 0.35 seconds. This

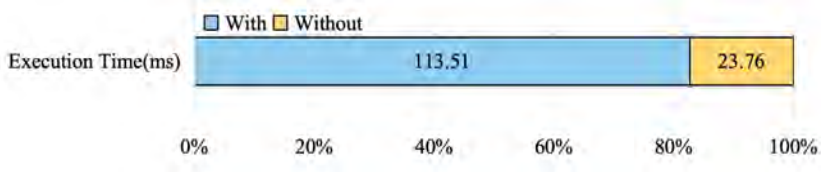

Fig. 8. The effects of call edges refinement.

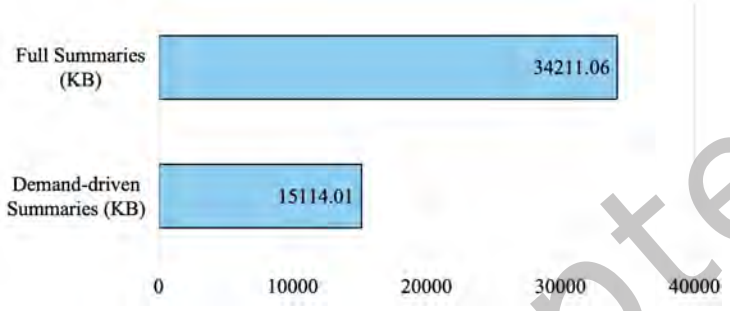

Fig. 9. The total storage usage for analyzing all projects.

showcases the significant potential of *CVERiskDetect* in accelerating existing detection tools, thereby meeting the requirements of the IDE environment.

Answer to RQ3: ReachCheck can significantly improve the efficiency of the two client analyses of call graph reachability analysis, dependency conflict detection and CVE risk detection, demonstrating its potential in the applications within IDE environments.

7 DISCUSSION

7.1 Impact of Call Edges Refinement

The call edge refinement optimization strategy within TPL call graphs in our approach (See Section 4.1) significantly enhances computational efficiency. This strategy involves selectively computing only the necessary call edges, starting with internal call graphs and subsequently updating summaries for outer call edges. Such selective computation drastically cuts down on unnecessary processing, thereby accelerating overall execution times and improving the system's capacity to handle large datasets efficiently. Figure 8 demonstrates the effectiveness of this approach, showing that after implementing call edge refinement, the average execution time for the TC algorithm on the GPU decreased significantly by 79.07%. This substantial reduction underscores the effectiveness of the strategy in optimizing the performance and resource utilization.

7.2 Impact of Demand-driven Summary Retrieval

The strategy of demand-driven summary retrieval is designed to streamline the querying process by simplifying data retrieval for TPLs. We can compute the call edges between the intermediate node $L_t \subseteq D_p$ and the TPLs it depends on using only the outer call graph summary S_o. This method enables us to omit the call edges of its inner

Table 3. Github projects which invoke vulnerable APIs

Project Name	Bug ID	Stars/Forks	CVE ID	Status
apollo	#4755	28.1k/10.1k	2022-25857	Confirmed
hutool	#2999	26.4k/7.1k	2022-25857	Fixed
openapi-generator	#15195	16.6k/5.3k	2018-12537	Fixed
sofa-jraft	#960	3.2k/1k	2022-25857	Fixed
webcam-capture	#879	2.1k/1.1k	2020-13956	Fixed
inlong	#7480	1.2k/420	2019-17563	Fixed
datagear	#23	931/275	2022-31692	Fixed
rsocket-broker	#220	722/156	2022-25857	Fixed
mendmix	#17	664/289	2022-25857	Fixed
wcm-components	#2358	648/699	2020-9484	Fixed
jinjava	#1049	601/151	2022-25857	Confirmed
infinitest	#351	572/152	2020-15250	Fixed
Web-Karma	#573	557/196	2020-13956	Fixed
vertx	#95	544/191	2019-17640	Confirmed
query-federation	#829	498/258	2020-13956	Fixed
congomall	#18	442/55	2022-25857	Fixed
camellia	#98	356/90	2018-1000873	Fixed
core-geonetwork	#6759	354/452	2020-13956	Fixed
NCM2MP3	#9	278/54	2022-25845	Fixed
OpenAudioMc	#306	267/90	2022-25857	Fixed
rumble	#1226	194/78	2022-25857	Fixed
h2gis	#1338	191/65	2022-21724	Fixed
h2gis	#1338	191/65	2022-26520	Fixed
JedAIToolkit	#63	191/39	2020-13956	Fixed
ruoyi	#1	176/35	2021-36090	Fixed
scblogs	#169	169/50	2022-25857	Fixed
FastBeeIM	#2	141/39	2022-25857	Fixed
rocketmq-connect	#434	90/99	2022-25845	Fixed
BlackLab	#419	85/51	2022-25857	Fixed
Stitching	#70	83/61	2022-25857	Fixed
kungfu	#2	81/61	2022-25845	Fixed
californium.tools	#86	59/56	2017-7656	Fixed
jdcloud-sdk-java	#274	37/41	2020-13956	Fixed
alcor	#760	29/34	2020-13956	Fixed
helidon-build-tools	#850	29/34	2020-13956	Fixed
unix-monitor	#55	27/18	2020-13956	Fixed
hugegraph-commons	#109	26/37	2020-15250	Fixed
gestionmateriel	#39	23/38	2022-25857	Fixed
basyx-java-sdk	#276	22/27	2021-36090	Fixed
elasticjob	#2153	7.9k/3.3k	2020-13956	Fixed
error-prone	#3835	6.5k/746	2022-25857	Fixed
MoonBox	#27	769/126	2022-22970	Fixed
restx	#339	440/80	2017-12615	Fixed

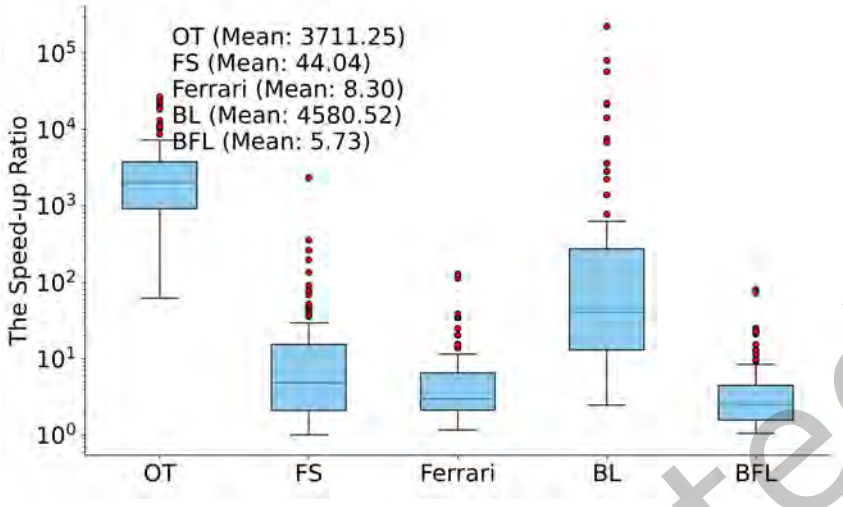

Fig. 10. The speedup compare to other approaches (VTA call graph).

methods, resulting in significant space savings. Figure 9 shows that after demand-driven summary retrieving, the total storage needed was reduced by 55.82%, demonstrating substantial efficiency gains in large-scale systems.

7.3 Time and Space Cost of Summary Computation

The computation of the TC typically requires substantial processing time. To optimize this, we leverage GPUs for fast matrix multiplication. Specifically, we use cuBool [44] on the GPU, which reduces the TC computation time to the millisecond range. Our experiments show that generating summaries for each TPL used in the *ReachBench* projects takes an average of only 23.76 ms. The summary construction is designed as an offline preprocessing step and is efficiently accelerated using GPU computation, making it practical even at scale. On average, it consumes only around 272.72 MB of GPU memory per TPL, which indicates a low GPU memory overhead. For example, the popular TPL `httpclient-4.5.1`, which includes 11,632 direct call edges and 4,616 methods, has its TC computed in just 9.6 ms using 271.56 MB of GPU memory. This efficiency enables us to precompute the TC of all Maven TPLs offline on the server. The client plugin can then retrieve and reuse these precomputed results for rapid reachability analysis, significantly accelerating detection.

In terms of space, the optimizations in the summary structure have reduced the storage requirement for each TPL's summary to just 1.81% of the original TPL binary file. For example, the summary for the TPL `httpclient-4.5.1` occupies only 3.65% of the original JAR file size. This low storage cost makes it feasible to deploy these summaries alongside TPLs in Maven repositories in the future. IDEs can then download and use these summaries directly, enabling efficient call graph reachability queries.

7.4 Impact of Call Graph Construction Algorithms

To assess whether the effectiveness of ReachCheck is influenced by the choice of call graph construction algorithms, we conducted an additional experiment using VTA, a more precise algorithm supported by Soot. Compared to the CHA based construction used in our main experiments, VTA performs more refined resolution of dynamic dispatch, resulting in more precise call graphs.

After regenerating the call graphs for all benchmark projects using VTA, we observed that the average degree decreased from 5.13 to 3.80, indicating a 25.9% reduction in graph density. To verify whether ReachCheck's performance advantage holds under this new setting, we re-evaluated ReachCheck and all baseline techniques using the VTA-generated call graphs, following the same experimental setup as described in Section 6.3. We measured only the query execution time for all baselines. The results show that ReachCheck continues to significantly outperform all baselines. Specifically, as shown in Figure 10, when using the VTA-based call graph, ReachCheck achieves a speedup of 3, 711.25× over the OT approach, 44.04× over the FS approach, 8.30× over the Ferrari, 4, 580.52× over the BL and 5.73× over the BFL. Meanwhile, as shown in Figure 7, when using the CHA-based call graph, ReachCheck achieves a speedup of 237.75× over the OT approach, 78.55× over the FS approach, 84.86× over Ferrari, 4, 369.09× over BL, and 80.91× over BFL. The results show that adopting more precise call graph construction algorithms also can improve the query performance of ReachCheck.

7.5 Threats to Validity

The validity of our method hinges on environmental factors affecting computational efficiency. We mitigated this by conducting evaluations on a dedicated server, free from user interference and we repeated experiments to counteract caching effects. This ensured controlled conditions for reliable comparative analysis.

7.6 Limitations and Future Work

The limitations of static call graph analysis may affect the effectiveness of ReachCheck. In particular, language features such as Java reflection and other dynamic behaviors may lead to the unsoundness in static call graph construction, potentially missing actual call edges at runtime [62]. Recent research has proposed advanced techniques to improve soundness under such conditions [46, 57, 58]. These advancements are orthogonal to ReachCheck and can enhance the quality of underlying call graphs, thereby improving the effectiveness of downstream tasks such as vulnerability detection. While our current implementation uses CHA-based call graphs, Section 7.4 shows that adopting more precise algorithms such as VTA can result in sparser graphs and improve the performance. This illustrates that ReachCheck can naturally benefit from the improvements in static call graph construction while remaining agnostic to the specific algorithm used. As shown in Section 7.4, using the VTA algorithm reduces the average call graph sparsity from 5.13 to 3.80, which directly benefits our summary construction. For instance, in the case of the library `httpclient-4.5.1`, the construction time decreased from 9.6 ms to 8.6 ms, representing an improvement of approximately 10.42%. This demonstrates that higher call graph precision can lead to more efficient offline summary construction. With ongoing advancements in static analysis (e.g., class hierarchy analysis, rapid type analysis, and variable type analysis [10, 59]), both the soundness and precision of call edge construction are expected to improve, which will further enhance the efficiency and effectiveness of ReachCheck.

In future work, we plan to extend ReachCheck's summarization strategy beyond TPLs to also include stable parts of the application code, such as unchanged classes or packages. This extension would enable finer-grained incremental analysis within the IDE by avoiding redundant reanalysis of code that has not been modified.

8 RELATED WORK

8.1 Risk Detection in Ecosystem

Modern software development increasingly relies on TPLs, which are often vulnerable to various security threats [73]. Recent studies have thoroughly investigated these vulnerabilities across prominent software ecosystems like Java, JavaScript, and Python[39, 64, 82, 83]. Particularly, Wang et al.[67–69] proposed a tool to detect whether there are methods in the project that cause exceptions at runtime. These tools utilize online traversal algorithms for reachability analysis to identify risky method invocations within ecosystems. However, the significant time

costs of online traversal algorithms constrain their utility in practical applications. Our approach accelerates the computational process by operating orthogonally to existing methods, offering a more efficient solution for call graph reachability analysis.

8.2 IDE Detection Plugin

IDEs encompass a broad range of functionalities that significantly streamline developer tasks. Technologies like ECHO [79] and D4 [45] enhance IDE capabilities by optimizing error detection processes. ECHO optimizes perceptual analysis to distribute the analysis cost and avoid redundant computations, enabling it to detect vulnerabilities at the IDE level with increased speed. Concurrently, D4 achieves rapid error detection, effectively identifying issues within 100 milliseconds post-code modifications. This rapid detection is critical as it aligns with the time-sensitive requirements of modern development environments where long execution times can disrupt workflows and reduce the adoption rate of tools [34].

The design of our detection approach was initially motivated by the specific needs of IDE scenarios, aiming to seamlessly integrate with and enhance existing IDE detection capabilities. Experimental results confirm that our technology not only complements but also accelerates the performance of traditional IDE detection tools, providing developers with immediate feedback and significantly reducing the downtime associated with error correction.

8.3 Reachability Indices

Graph reachability queries, crucial for establishing if nodes within a graph can reach each other, are commonly processed using methods like DFS, BFS, or Arrow [55]. With the challenges posed by large graphs, several indexing methods have been developed to enhance query efficiency. These methods can be broadly categorized into three types: Tree-Cover Index [26, 30, 40, 56, 63, 65, 76, 77], 2-Hop Labeling Method [28, 29, 31, 32, 36, 41, 42, 47, 53, 75, 78, 84], and Approximate TC [61, 70, 71]. Tree-Cover Index assigns intervals $[a_v, b_v]$ to nodes, leveraging the spanning tree's structure to ascertain reachability [26]. It ensures comprehensive coverage by incorporating necessary adjustments for non-spreading edges. 2-Hop Labeling Method assigns two distinct label sets to each vertex, one representing vertices that are reachable from it and the other for vertices that can reach it, offering efficiency despite the NP-hard nature of constructing an optimal 2-hop index [32]. Approximate TC method reduces the size of the reachability set by applying a function, which transforms the set of reachable vertices from a given vertex into a smaller subset while ensuring no false negatives [71]. Special methods like DAGGER [77], IP [71], and DBL [47] provide dynamic index updates, though their speeds may not surpass traditional BFS [81].

Unlike graph index methods like the Tree-Cover Index, our approach utilizes an offline TC Index. This index pre-stores reachability information, eliminating the need for runtime recalculations. Particularly effective in static environments with infrequent graph updates, the offline TC Index offers immediate query responses and reduces overhead, ideal for use in IDEs where stability and efficiency are crucial.

8.4 Function Summaries in Compositional Analysis

Function summaries are widely used for caching the results of the compositional analysis. Various kinds of function summaries are designed to cope with different analyses. For example, function summaries in IFDS [52] and interprocedural distributive environment [60] frameworks are used for data-flow analysis. Arzt et al. [27] proposed flow-sensitive function summaries to support the taint analysis in Android. Wu et al. [72] design a two-layer function summaries to achieve path-sensitive analysis for detecting memory errors. Since the demands of the aforementioned client analysis are different, their function summaries are not feasible for tackling call graph reachability analysis. The most relevant work to our research goal [43] used the pre-analyzed call graph

as function summaries. However, due to the high time cost of online graph traversal, such function summaries are still unsatisfactory in the context of IDEs. Our proposed summary is different from the function summaries designed in prior study [43]. Specifically, our summary compactly encodes the reachability relation among APIs in a TPL and supports accelerating call graph reachability queries without online BFS or DFS graph traversal.

A closely related approach is presented by Schubert et al. [54], who compress control-flow graphs of individual programs to persist and reuse static analysis results, aiming to reduce recomputation overhead in traditional analysis pipelines. In contrast, our goal is to enable fast reachability queries within IDEs by significantly reducing the need for online graph traversals. ModAlyzer generates summaries that capture a broad range of control-flow features and therefore require more storage. In comparison, we precompute lightweight summaries for TPLs that compactly encode only call graph reachability between exported APIs. This focused and compact design makes our approach more suitable for interactive IDE environments, where low latency are critical. Meanwhile, Hejderup et al. [37, 38] construct versioned, ecosystem-wide call graphs across centralized repositories (e.g., Maven, NPM), supporting large-scale vulnerability propagation studies. Our work, however, targets project-specific, real-time analysis scenarios where developers frequently add or upgrade dependencies. Rather than building a global database, we generate modular summaries for each TPL, which can be dynamically composed at query time to reflect the current dependency state.

9 CONCLUSION

Call graph reachability analysis is a widely utilized technique in vulnerability detection, but the long detection times of existing accessibility analysis tools have limited their use in IDEs. This paper introduces ReachCheck, a novel compositional library-aware call graph reachability analysis that addresses the reachability queries of method pairs in the IDE. Experimental results demonstrate that ReachCheck efficiently constructs call graphs on-demand for TPLs, achieving an impressive average query time of just 105.17 ms per project. These advancements significantly enhance the overall efficiency of vulnerability detection in software projects and facilitate the practical application of detection tools within IDE environments. The source code of ReachCheck is available at https://github.com/ReachCheck/ReachCheck.

ACKNOWLEDGEMENT

We sincerely thank the anonymous reviewers for their valuable and insightful feedback. We also extend our gratitude to Qiao Xiang for his constructive suggestions on the manuscript. This research was supported by the National Key R&D Program of China (2022YFB2901502), the Natural Science Foundation of China (Grant No. 62272400), Fujian Provincial Natural Science Foundation of China (Grant No. 2025J010002), and the Leading-edge Technology Program of Jiangsu Natural Science Foundation (BK20202001). Rongxin Wu is the corresponding author and works as a member of Xiamen Key Laboratory of Intelligent Storage and Computing in Xiamen University.

REFERENCES

[1] 2018. Issue309. https://github.com/ff4j/ff4j/issues/309
[2] 2023. Dependency Analyzer. https://www.jetbrains.com/guide/java/tutorials/analyzing-dependencies/dependency-analyzer/
[3] 2023. Snyk. https://security.snyk.io/
[4] 2023. SootUp. https://github.com/soot-oss/SootUp
[5] 2024. CVE Reports. https://github.com/ReachCheck/ReachCheck/blob/main/reportData/CVE_reports.md
[6] 2024. DC Reports. https://github.com/ReachCheck/ReachCheck/blob/main/reportData/DC_reports.md
[7] 2024. IDEA. https://www.jetbrains.com/idea/
[8] 2024. Vulnerability Checker. https://www.jetbrains.com/help/idea/package-analysis.html
[9] 2025. BEAM-3690. https://issues.apache.org/jira/browse/BEAM-3690
[10] 2025. call-graph-construction. https://soot-oss.github.io/SootUp/v1.1.2/call-graph-construction/

[11] 2025. ff4j. https://github.com/ff4j/ff4j
[12] 2025. HADOOP-15261. https://issues.apache.org/jira/browse/HADOOP-15261
[13] 2025. Issue1371. https://github.com/locationtech/geowave/issues/1371
[14] 2025. Issue146. https://github.com/st-js/st-js/issues/146
[15] 2025. Issue2134. https://github.com/apache/dubbo/issues/2134
[16] 2025. Issue227. https://github.com/ExpediaGroup/styx/issues/227
[17] 2025. Issue272. https://github.com/jvelo/mayocat-shop/issues/272
[18] 2025. Issue315. https://github.com/ff4j/ff4j/issues/315
[19] 2025. Issue345. https://github.com/Azure/azure-storage-java/issues/345
[20] 2025. Issue39. https://github.com/subchen/jetbrick-template-2x/issues/39
[21] 2025. Issue473. https://github.com/google/truth/issues/473
[22] 2025. Issue477. https://github.com/vipshop/Saturn/issues/477
[23] 2025. Issue653. https://github.com/sarxos/webcam-capture/issues/653
[24] 2025. ReachBench. https://github.com/ReachCheck/ReachCheck/blob/main/Data/ReachBench.md
[25] 2025. Storm-3171. https://issues.apache.org/jira/browse/STORM-3171
[26] Rakesh Agrawal, Alexander Borgida, and H. V. Jagadish. 1989. Efficient Management of Transitive Relationships in Large Data and Knowledge Bases. In *Proceedings of the 1989 ACM SIGMOD International Conference on Management of Data, Portland, Oregon, USA, May 31 - June 2, 1989*, James Clifford, Bruce G. Lindsay, and David Maier (Eds.). ACM Press, 253–262. https://doi.org/10.1145/67544.66950
[27] Steven Arzt and Eric Bodden. 2016. StubDroid: Automatic inference of precise data-flow summaries for the Android framework. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 725–735.
[28] Ramadhana Bramandia, Byron Choi, and Wee Keong Ng. 2010. Incremental Maintenance of 2-Hop Labeling of Large Graphs. *IEEE Trans. Knowl. Data Eng.* 22, 5 (2010), 682–698. https://doi.org/10.1109/TKDE.2009.117
[29] Jing Cai and Chung Keung Poon. 2010. Path-hop: efficiently indexing large graphs for reachability queries. In *Proceedings of the 19th ACM Conference on Information and Knowledge Management, CIKM 2010, Toronto, Ontario, Canada, October 26-30, 2010*, Jimmy X. Huang, Nick Koudas, Gareth J. F. Jones, Xindong Wu, Kevyn Collins-Thompson, and Aijun An (Eds.). ACM, 119–128. https://doi.org/10.1145/1871437.1871457
[30] Li Chen, Amarnath Gupta, and M. Erdem Kurul. 2005. Stack-based Algorithms for Pattern Matching on DAGs. In *Proceedings of the 31st International Conference on Very Large Data Bases, Trondheim, Norway, August 30 - September 2, 2005*, Klemens Böhm, Christian S. Jensen, Laura M. Haas, Martin L. Kersten, Per-Åke Larson, and Beng Chin Ooi (Eds.). ACM, 493–504. http://www.vldb.org/archives/website/2005/program/paper/wed/p493-chen.pdf
[31] James Cheng, Silu Huang, Huanhuan Wu, and Ada Wai-Chee Fu. 2013. TF-Label: a topological-folding labeling scheme for reachability querying in a large graph. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2013, New York, NY, USA, June 22-27, 2013*, Kenneth A. Ross, Divesh Srivastava, and Dimitris Papadias (Eds.). ACM, 193–204. https://doi.org/10.1145/2463676.2465286
[32] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and Distance Queries via 2-Hop Labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355. https://doi.org/10.1137/S0097539702403098
[33] Andreas Dann, Ben Hermann, and Eric Bodden. 2023. UPCY: Safely Updating Outdated Dependencies. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 233–244. https://doi.org/10.1109/ICSE48619.2023.00031
[34] Lisa Nguyen Quang Do, Karim Ali, Benjamin Livshits, Eric Bodden, Justin Smith, and Emerson R. Murphy-Hill. 2017. Cheetah: just-in-time taint analysis for Android apps. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017 - Companion Volume*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE Computer Society, 39–42. https://doi.org/10.1109/ICSE-C.2017.20
[35] Georgios-Petros Drosos, Thodoris Sotiropoulos, Diomidis Spinellis, and Dimitris Mitropoulos. 2024. Bloat beneath Python's Scales: A Fine-Grained Inter-Project Dependency Analysis. *Proc. ACM Softw. Eng.* 1, FSE, Article 114 (jul 2024), 24 pages. https://doi.org/10.1145/3660821
[36] Kathrin Hanauer, Christian Schulz, and Jonathan Trummer. 2022. O'Reach: Even Faster Reachability in Large Graphs. *ACM J. Exp. Algorithmics* 27 (2022), 4.2:1–4.2:27. https://doi.org/10.1145/3556540
[37] Joseph Hejderup, Moritz Beller, Konstantinos Triantafyllou, and Georgios Gousios. 2022. Präzi: from package-based to call-based dependency networks. *Empirical Software Engineering* 27, 5 (2022), 102.
[38] Joseph Hejderup, Arie Van Deursen, and Georgios Gousios. 2018. Software ecosystem call graph for dependency management. In *Proceedings of the 40th International Conference on Software Engineering: New Ideas and Emerging Results*. 101–104.
[39] Kaifeng Huang, Bihuan Chen, Congying Xu, Ying Wang, Bowen Shi, Xin Peng, Yijian Wu, and Yang Liu. 2022. Characterizing usages, updates and risks of third-party libraries in Java projects. *Empir. Softw. Eng.* 27, 4 (2022), 90. https://doi.org/10.1007/S10664-022-10131-8
[40] Ruoming Jin, Ning Ruan, Yang Xiang, and Haixun Wang. 2011. Path-tree: An efficient reachability indexing scheme for large directed graphs. *ACM Trans. Database Syst.* 36, 1 (2011), 7:1–7:44. https://doi.org/10.1145/1929934.1929941

[41] Ruoming Jin and Guan Wang. 2013. Simple, Fast, and Scalable Reachability Oracle. *Proc. VLDB Endow.* 6, 14 (2013), 1978–1989. https://doi.org/10.14778/2556549.2556578

[42] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 2009. 3-HOP: a high-compression indexing scheme for reachability query. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, SIGMOD 2009, Providence, Rhode Island, USA, June 29 - July 2, 2009*, Ugur Çetintemel, Stanley B. Zdonik, Donald Kossmann, and Nesime Tatbul (Eds.). ACM, 813–826. https://doi.org/10.1145/1559845.1559930

[43] Mehdi Keshani, Georgios Gousios, and Sebastian Proksch. 2024. Frankenstein: fast and lightweight call graph generation for software builds. *Empir. Softw. Eng.* 29, 1 (2024), 1. https://doi.org/10.1007/S10664-023-10388-7

[44] Robin Kobus, Adrian Lamoth, André Müller, Christian Hundt, Stefan Kramer, and Bertil Schmidt. 2018. cuBool: Bit-Parallel Boolean Matrix Factorization on CUDA-Enabled Accelerators. In *24th IEEE International Conference on Parallel and Distributed Systems, ICPADS 2018, Singapore, December 11-13, 2018*. IEEE, 465–472. https://doi.org/10.1109/PADSW.2018.8644574

[45] Bozhen Liu and Jeff Huang. 2018. D4: fast concurrency debugging with parallel differential analysis. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, Jeffrey S. Foster and Dan Grossman (Eds.). ACM, 359–373. https://doi.org/10.1145/3192366.3192390

[46] Benjamin Livshits, John Whaley, and Monica S. Lam. 2005. Reflection analysis for java. In *Proceedings of the Third Asian Conference on Programming Languages and Systems* (Tsukuba, Japan) *(APLAS'05)*. Springer-Verlag, Berlin, Heidelberg, 139–160. https://doi.org/10.1007/11575467_11

[47] Qiuyi Lyu, Yuchen Li, Bingsheng He, and Bin Gong. 2021. DBL: Efficient Reachability Queries on Dynamic Graphs. In *Database Systems for Advanced Applications - 26th International Conference, DASFAA 2021, Taipei, Taiwan, April 11-14, 2021, Proceedings, Part II (Lecture Notes in Computer Science, Vol. 12682)*, Christian S. Jensen, Ee-Peng Lim, De-Nian Yang, Wang-Chien Lee, Vincent S. Tseng, Vana Kalogeraki, Jen-Wei Huang, and Chih-Ya Shen (Eds.). Springer, 761–777. https://doi.org/10.1007/978-3-030-73197-7_52

[48] Egor Orachev, Maria Karpenko, Artem Khoroshev, and Semyon V. Grigorev. 2021. SPbLA: The Library of GPGPU-Powered Sparse Boolean Linear Algebra Operations. In *IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2021, Portland, OR, USA, June 17-21, 2021*. IEEE, 272–275. https://doi.org/10.1109/IPDPSW52791.2021.00049

[49] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2018. 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 2018, Madrid, Spain, September 23-29, 2018*. IEEE Computer Society, 449–460. https://doi.org/10.1109/ICSME.2018.00054

[50] Serena Elisa Ponta, Henrik Plate, and Antonino Sabetta. 2020. Detection, assessment and mitigation of vulnerabilities in open source dependencies. *Empir. Softw. Eng.* 25, 5 (2020), 3175–3215. https://doi.org/10.1007/S10664-020-09830-X

[51] Michael Reif, Michael Eichberg, Ben Hermann, Johannes Lerch, and Mira Mezini. 2016. Call graph construction for Java libraries. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 474–486. https://doi.org/10.1145/2950290.2950312

[52] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 49–61.

[53] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. 2005. Efficient Creation and Incremental Maintenance of the HOPI Index for Complex XML Document Collections. In *Proceedings of the 21st International Conference on Data Engineering, ICDE 2005, 5-8 April 2005, Tokyo, Japan*, Karl Aberer, Michael J. Franklin, and Shojiro Nishio (Eds.). IEEE Computer Society, 360–371. https://doi.org/10.1109/ICDE.2005.57

[54] Philipp Dominik Schubert, Ben Hermann, and Eric Bodden. 2021. Lossless, persisted summarization of static callgraph, points-to and data-flow analysis. In *35th European Conference on Object-Oriented Programming (ECOOP 2021)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2–1.

[55] Neha Sengupta, Amitabha Bagchi, Maya Ramanath, and Srikanta Bedathur. 2019. ARROW: Approximating Reachability Using Random Walks Over Web-Scale Graphs. In *35th IEEE International Conference on Data Engineering, ICDE 2019, Macao, China, April 8-11, 2019*. IEEE, 470–481. https://doi.org/10.1109/ICDE.2019.00049

[56] Stephan Seufert, Avishek Anand, Srikanta J. Bedathur, and Gerhard Weikum. 2013. FERRARI: Flexible and efficient reachability range assignment for graph indexing. In *29th IEEE International Conference on Data Engineering, ICDE 2013, Brisbane, Australia, April 8-12, 2013*, Christian S. Jensen, Christopher M. Jermaine, and Xiaofang Zhou (Eds.). IEEE Computer Society, 1009–1020. https://doi.org/10.1109/ICDE.2013.6544893

[57] Yannis Smaragdakis, George Balatsouras, George Kastrinis, and Martin Bravenboer. 2015. More sound static handling of Java reflection. In *Asian Symposium on Programming Languages and Systems*. Springer, 485–503.

[58] Xiaohu Song, Ying Wang, Xiao Cheng, Guangtai Liang, Qianxiang Wang, and Zhiliang Zhu. 2024. Efficiently Trimming the Fat: Streamlining Software Dependencies with Java Reflection and Dependency Analysis. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 103:1–103:12. https://doi.org/10.1145/3597503.3639123

[59] Xiaohu Song, Ying Wang, Xiao Cheng, Guangtai Liang, Qianxiang Wang, and Zhiliang Zhu. 2024. Efficiently Trimming the Fat: Streamlining Software Dependencies with Java Reflection and Dependency Analysis. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering* (Lisbon, Portugal) *(ICSE '24)*. Association for Computing Machinery, New York, NY, USA, Article 103, 12 pages. https://doi.org/10.1145/3597503.3639123

[60] Johannes Späth, Karim Ali, and Eric Bodden. 2017. Ide al: Efficient and precise alias-aware dataflow analysis. *Proceedings of the ACM on Programming Languages* 1, OOPSLA (2017), 1–27.

[61] Jiao Su, Qing Zhu, Hao Wei, and Jeffrey Xu Yu. 2017. Reachability Querying: Can It Be Even Faster? *IEEE Trans. Knowl. Data Eng.* 29, 3 (2017), 683–697. https://doi.org/10.1109/TKDE.2016.2631160

[62] Li Sui, Jens Dietrich, Amjed Tahir, and George Fourtounis. 2020. On the recall of static call graph construction in practice. In *ICSE '20: 42nd International Conference on Software Engineering, Seoul, South Korea, 27 June - 19 July, 2020*, Gregg Rothermel and Doo-Hwan Bae (Eds.). ACM, 1049–1060. https://doi.org/10.1145/3377811.3380441

[63] Silke Trißl and Ulf Leser. 2007. Fast and practical indexing and querying of very large graphs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data, Beijing, China, June 12-14, 2007*, Chee Yong Chan, Beng Chin Ooi, and Aoying Zhou (Eds.). ACM, 845–856. https://doi.org/10.1145/1247480.1247573

[64] Chao Wang, Rongxin Wu, Haohao Song, Jiwu Shu, and Guoqing Li. 2022. smartPip: A Smart Approach to Resolving Python Dependency Conflict Issues. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 93:1–93:12. https://doi.org/10.1145/3551349.3560437

[65] Haixun Wang, Hao He, Jun Yang, Philip S. Yu, and Jeffrey Xu Yu. 2006. Dual Labeling: Answering Graph Reachability Queries in Constant Time. In *Proceedings of the 22nd International Conference on Data Engineering, ICDE 2006, 3-8 April 2006, Atlanta, GA, USA*, Ling Liu, Andreas Reuter, Kyu-Young Whang, and Jianjun Zhang (Eds.). IEEE Computer Society, 75. https://doi.org/10.1109/ICDE.2006.53

[66] Huiyan Wang, Shuguan Liu, Lingyu Zhang, and Chang Xu. 2023. Automatically Resolving Dependency-Conflict Building Failures via Behavior-Consistent Loosening of Library Version Constraints. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 198–210. https://doi.org/10.1145/3611643.3616264

[67] Ying Wang, Ming Wen, Zhenwei Liu, Rongxin Wu, Rui Wang, Bo Yang, Hai Yu, Zhiliang Zhu, and Shing-Chi Cheung. 2018. Do the dependency conflicts in my project matter?. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2018, Lake Buena Vista, FL, USA, November 04-09, 2018*, Gary T. Leavens, Alessandro Garcia, and Corina S. Pasareanu (Eds.). ACM, 319–330. https://doi.org/10.1145/3236024.3236056

[68] Ying Wang, Ming Wen, Rongxin Wu, Zhenwei Liu, Shin Hwei Tan, Zhiliang Zhu, Hai Yu, and Shing-Chi Cheung. 2019. Could I have a stack trace to examine the dependency conflict issue?. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, 572–583. https://doi.org/10.1109/ICSE.2019.00068

[69] Ying Wang, Rongxin Wu, Chao Wang, Ming Wen, Yepang Liu, Shing-Chi Cheung, Hai Yu, Chang Xu, and Zhiliang Zhu. 2022. Will Dependency Conflicts Affect My Program's Semantics? *IEEE Trans. Software Eng.* 48, 7 (2022), 2295–2316. https://doi.org/10.1109/TSE.2021.3057767

[70] Hao Wei, Jeffrey Xu Yu, Can Lu, and Ruoming Jin. 2014. Reachability Querying: An Independent Permutation Labeling Approach. *Proc. VLDB Endow.* 7, 12 (2014), 1191–1202. https://doi.org/10.14778/2732977.2732992

[71] Hao Wei, Jeffrey Xu Yu, Can Lu, and Ruoming Jin. 2018. Reachability querying: an independent permutation labeling approach. *VLDB J.* 27, 1 (2018), 1–26. https://doi.org/10.1007/S00778-017-0468-3

[72] Rongxin Wu, Yuxuan He, Jiafeng Huang, Chengpeng Wang, Wensheng Tang, Qingkai Shi, Xiao Xiao, and Charles Zhang. 2024. LibAlchemy: A Two-Layer Persistent Summary Design for Taming Third-Party Libraries in Static Bug-Finding Systems. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*. ACM, 105:1–105:13. https://doi.org/10.1145/3597503.3639132

[73] Yulun Wu, Zeliang Yu, Ming Wen, Qiang Li, Deqing Zou, and Hai Jin. 2023. Understanding the Threats of Upstream Vulnerabilities to Downstream Projects in the Maven Ecosystem. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 1046–1058. https://doi.org/10.1109/ICSE48619.2023.00095

[74] Meiqiu Xu, Ying Wang, Shing-Chi Cheung, Hai Yu, and Zhiliang Zhu. 2022. Insight: Exploring Cross-Ecosystem Vulnerability Impacts. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 58:1–58:13. https://doi.org/10.1145/3551349.3556921

[75] Yosuke Yano, Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast and scalable reachability queries on graphs by pruned labeling with landmarks and paths. In *22nd ACM International Conference on Information and Knowledge Management, CIKM'13, San Francisco, CA, USA, October 27 - November 1, 2013*, Qi He, Arun Iyengar, Wolfgang Nejdl, Jian Pei, and Rajeev Rastogi (Eds.). ACM, 1601–1606. https://doi.org/10.1145/2505515.2505724

[76] Hilmi Yildirim, Vineet Chaoji, and Mohammed Javeed Zaki. 2010. GRAIL: Scalable Reachability Index for Large Graphs. *Proc. VLDB Endow.* 3, 1 (2010), 276–284. https://doi.org/10.14778/1920841.1920879

[77] Hilmi Yildirim, Vineet Chaoji, and Mohammed J. Zaki. 2013. DAGGER: A Scalable Index for Reachability Queries in Large Dynamic Graphs. *CoRR* abs/1301.0977 (2013). arXiv:1301.0977 http://arxiv.org/abs/1301.0977

[78] Changyong Yu, Tianmei Ren, Wenyu Li, Huimin Liu, Haitao Ma, and Yuhai Zhao. 2024. BL: An Efficient Index for Reachability Queries on Large Graphs. *IEEE Trans. Big Data* 10, 2 (2024), 108–121. https://doi.org/10.1109/TBDATA.2023.3327215

[79] Sheng Zhan and Jeff Huang. 2016. ECHO: instantaneous in situ race detection in the IDE. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, Thomas Zimmermann, Jane Cleland-Huang, and Zhendong Su (Eds.). ACM, 775–786. https://doi.org/10.1145/2950290.2950332

[80] Chao Zhang, Angela Bonifati, and M. Tamer Özsu. 2023. An Overview of Reachability Indexes on Graphs. In *Companion of the 2023 International Conference on Management of Data, SIGMOD/PODS 2023, Seattle, WA, USA, June 18-23, 2023*, Sudipto Das, Ippokratis Pandis, K. Selçuk Candan, and Sihem Amer-Yahia (Eds.). ACM, 61–68. https://doi.org/10.1145/3555041.3589408

[81] Chao Zhang, Angela Bonifati, and M. Tamer Özsu. 2023. An Overview of Reachability Indexes on Graphs. In *Companion of the 2023 International Conference on Management of Data* (Seattle, WA, USA) *(SIGMOD '23)*. Association for Computing Machinery, New York, NY, USA, 61–68. https://doi.org/10.1145/3555041.3589408

[82] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Bihuan Chen, and Yang Liu. 2022. Has My Release Disobeyed Semantic Versioning? Static Detection Based on Semantic Differencing. In *37th IEEE/ACM International Conference on Automated Software Engineering, ASE 2022, Rochester, MI, USA, October 10-14, 2022*. ACM, 51:1–51:12. https://doi.org/10.1145/3551349.3556956

[83] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Lida Zhao, Jiahui Wu, and Yang Liu. 2023. Compatible Remediation on Vulnerabilities from Third-Party Libraries for Java Projects. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*. IEEE, 2540–2552. https://doi.org/10.1109/ICSE48619.2023.00212

[84] Andy Diwen Zhu, Wenqing Lin, Sibo Wang, and Xiaokui Xiao. 2014. Reachability queries on large dynamic graphs: a total order approach. In *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, Curtis E. Dyreson, Feifei Li, and M. Tamer Özsu (Eds.). ACM, 1323–1334. https://doi.org/10.1145/2588555.2612181