Compatible Remediation on Vulnerabilities from Third-Party Libraries for Java Projects

Lyuye Zhang*, Chengwei Liu*§, Zhengzi Xu*, Sen Chen†§, Lingling Fan‡, Lida Zhao*, Jiahui Wu*, Yang Liu*
*School of Computer Science and Engineering, Nanyang Technological University, Singapore
†College of Intelligence and Computing, Tianjin University, China
‡College of Cyber Science, Nankai University, China

***Abstract*—With the increasing disclosure of vulnerabilities in open-source software, software composition analysis (SCA) has been widely applied to reveal third-party libraries and the associated vulnerabilities in software projects. Beyond the revelation, SCA tools adopt various remediation strategies to fix vulnerabilities, the quality of which varies substantially. However, ineffective remediation could induce side effects, such as compilation failures, which impede acceptance by users. According to our studies, existing SCA tools could not correctly handle the concerns of users regarding the compatibility of remediated projects. To this end, we propose Compatible Remediation of Third-party libraries (CORAL) for Maven projects to fix vulnerabilities without breaking the projects. The evaluation proved that CORAL not only fixed 87.56% of vulnerabilities which outperformed other tools (best 75.32%) and achieved a 98.67% successful compilation rate and a 92.96% successful unit test rate. Furthermore, we found that 78.45% of vulnerabilities in popular Maven projects could be fixed without breaking the compilation, and the rest of the vulnerabilities (21.55%) could either be fixed by upgrades that break the compilations or even be impossible to fix by upgrading.**

***Index Terms*—Remediation, Compatibility, Java, Open-source software**

I. INTRODUCTION

The exposure of open-source third-party libraries (TPLs) vulnerabilities in recent years, such as the well-known Log4Shell vulnerability [1], [2], has been drawing increasing attention. To accurately detect the versioned TPLs and the disclosed vulnerabilities in users' projects, software composition analysis (SCA) [3] has been widely applied to scan projects and return detected TPLs for security analysis. The detection has been well developed and implemented in various academic and commercial SCA tools [4]–[9]. However, the remediation to fix vulnerabilities in TPLs by version adjustments has no broadly acknowledged solution but various strategies.

We further investigated existing tools. Community tools, such as Dependabot [5], only considers vulnerabilities of direct dependencies. Other popular anonymous commercial tools use reachability analysis as the prioritization metric, but none considers the compatibility of upgrades of the dependencies. An academic tool, Steady, calculates the percentage of changed classes or methods as the probability for compatibility, which is inaccurate by its nature of uncertainty.

§ Chengwei Liu and Sen Chen are the corresponding authors (Emails: chengwei001@e.ntu.edu.sg, senchen@tju.edu.cn).

Due to different strategies, the effectiveness of remediation tools varies substantially, which will be clarified in the preliminary study and the evaluation. Moreover, the side effects of remediation could hinder the adoption of suggestions by users. According to our study [10] of the rejected remediation suggestions at GitHub, the primary concern of users was incompatibility, which accounted for 51.31%.

Unfortunately, these concerns, especially compatibility, cannot be appropriately handled by existing remediation tools due to two reasons: (1) They conduct local optimization on individual libraries instead of the global optimization of the entire dependency graph (DG), which may miss incompatible relationships and fail to handle the trade-off between compatibility and security. (2) They offer suggestions based on the original DG and overlook the structural changes that suggestions bring to it and the underlying call graphs. As a result, the outdated DG could lead to incompatibility, lack of remediation on new vulnerabilities, and wasted remediation on unused dependencies.

To address the problems of existing tools and achieve remediation of better quality, three major **challenges** have to be resolved: **c1**: The absolutely optimal solutions for libraries are not always available. So the trade-off between security and compatibility during decision-making has to be handled. **c2**: The complexity of global optimization increases exponentially with the number of dependencies because the version combinations over all libraries should be traversed. **c3**: The suggestions on one library can, directly and indirectly, change the DG structure, call graphs, and compatibility of DG. Accordingly, the optimal solutions for the rest of the libraries may also be changed. These effects propagate from the changed library to the entire DG through dependency relationships, referred to as *ripple effects* in this paper. The *ripple effects* may lead to sub-optimal solutions if DG is not updated accordingly.

To tackle the above-mentioned challenges, we propose Compatible Remediation of Third-party Libraries (CORAL) to remediate vulnerabilities in dependencies by version suggestions without breaking the projects with a balanced time cost for Maven [11]. CORAL starts with the DG and the underlying call graphs of the target project. Then, CORAL splits the DG into subgraphs with two steps of partitioning for **c1**. CORAL walkthroughs subgraphs with a top-down approach and calculates the best solutions with SMT solver

1558-1225/23/$31.00 ©2023 IEEE
DOI 10.1109/ICSE48619.2023.00212

within subgraphs for **c2**. During the walkthrough, subsequent subgraphs are dynamically updated for the *ripple effects* to handle **c3**. To avoid dead ends, backtracking mechanisms are implemented in CORAL.

We have evaluated CORAL by comparing it with state-of-the-art remediation tools regarding security and compatibility. It turned out CORAL fixed the most vulnerabilities (87.56%) among all tools (best of others 75.32%) and achieved the best assurance of compatibility (98.67% successful compilation rate and 92.96% successful unit test rate). Moreover, the designs of subgraph partitioning and the trade-off between compatibility and security were evaluated against the baselines. The result showed CORAL broke fewer projects and spent much less time than them at the cost of 4.05% fewer vulnerabilities fixed. Furthermore, we found that 78.45% of vulnerabilities in popular Maven projects could be fixed without breaking the projects. However, without the aid of CORAL, only 25.71% could be straightforwardly fixed by users. The contributions we have made are as follows.

- We proposed CORAL as a remediation tool for Maven projects to handle the global optimization for enhanced security and compatibility.
- We studied the concerns of users regarding remediation suggestions by analyzing Pull Requests (PRs) and found that 51.31% of cases were related to incompatibility.
- We empirically compared and analyzed strategies of popular remediation tools regarding their support of compatibility and prioritization for the reference of other researchers.

II. MOTIVATIONS

A. Motivating Example

Dependabot has been widely used as the most popular dependency security management extension at GitHub. One of the most popular Maven projects, *commons-lang* [12], adopted Dependabot to manage their dependencies. Nevertheless, the remediation caused build failure after upgrading [13]. Dependabot has implemented the compatibility score by calculating the test passing rates from other repositories as the confidence score. However, in this case, the compatibility score was *unknown*. The compatibility score relying on knowledge of the crowd cannot guarantee a successful compilation without code-based compatibility calculation. Thus, CORAL relies on static code-based compatibility checkers aligning with a global perspective of DG to ensure the adjusted dependencies do not break the project. Motivated by the motivating example, we studied the strategies of state-of-the-art remediation tools to understand how existing tools handle incompatibility issues. Then, we further studied the concerns of users regarding the remediation suggestions at GitHub to recognize what can be improved.

B. Study of Remediation Strategies of Existing Tools

To understand the implicit reasons for the breaking in Section II-A, we first empirically compared the published remediation strategies of existing tools and then quantitatively evaluated them in Section IV. We only counted tools that provided actionable advice for dependencies, while tools that only offered multiple suggestions for vulnerabilities were out of the discussion because users would have to select the version out of multiple suggestions manually during decision-making for each library. The tools included Dependabot, Steady, and two popular commercial tools denoted by *Com A* and *Com B*.

- **Dependabot**: Dependabot is able to create PRs to upgrade vulnerable dependencies to clean versions instead of providing an overall suggestion for the entire DG. As for the compatibility, Dependabot calculates the successful test rate of the upgrades from other repositories as the confidence score. However, this score can be unreliable because it is usually unavailable, and the compatibility ultimately depends on the context of the code base.
- **Steady**: Steady is an open-source academic SCA tool with an open-source vulnerability database. Steady adjusts the versions of both direct and transitive dependencies to reduce the vulnerability risks at a fine granularity. Also, it utilizes the reachability analysis of vulnerabilities to filter out the unreachable CVEs with low risks. The reachability comprises both static and dynamic analysis, which only constructs call graphs once at the beginning. As for the version selection, Steady prioritizes the non-vulnerable versions, then determines the best candidate with the compatibility probability p. To derive p, it defines the reachable constructs (class, method, etc.) as *touch points* and calculates the percentage of present *touch points* in upgraded versions as p. The probability could be unreliable due to its uncertainty.
- **Com A**: Towards a DG, Com A tweaks only the direct dependencies to remediate the vulnerabilities. The fundamental strategy is to upgrade the libraries with vulnerabilities to the closest non-vulnerable versions, as the closer versions usually are more likely to be compatible. The reachability is implemented by WALA [14] in a static manner to prioritize the critical reachable vulnerabilities. However, the compatibility of the remediation is not taken into account.
- **Com B**: Com B conducts the remediation on the direct dependencies. The key feature is that Com B considers all vulnerabilities of transitive dependencies associated with the direct dependencies. Specifically, it iterates over all direct dependencies. For each, Com B attempts the version candidates and resolves the subsequent dependencies to measure the updated overall vulnerabilities. Then, Com B selects the version with the fewest overall vulnerabilities for this direct dependency. The strategy considers the *ripple effects* from the upgraded direct dependency to the upstream tree. However, as direct dependencies are usually not independent but inter-connected by transitive dependency relationships, the respective optimization of each direct dependency does not necessarily result in global optimization.

The comparison of SCA tools is demonstrated in Table I. *Fix level* refers to the direct/transitive dependencies to be fixed. *Fix unit* denotes the basic units that the tools optimize. *S&C trade-off* means the prioritization of determining the best

TABLE I: Comparison of State-of-the-art SCA Tools That Provide Remediation

Tool	Fix level	Fix target	Compatibility	S & C trade-off	Reachability	Dep conflict	Ripple effects	Unused dependencies
Steady	All graph	Vertex	◐	Sec first	●	○	○	○
Dependabot	Direct	Vertex	◐	Sec first	○	○	○	○
Com A	Direct	Vertex	○	Sec only	●	○	○	○
Com B	Direct	Tree	○	Sec only	○	○	◐	○

1) *Fix level*: direct/direct+transitive dependencies. *Fix target*: basic units that the tools consider during optimization. *S&C trade-off*: prioritization of security or compatibility during version determination. *Ripple effects*: the support to handle the side effects brought by *ripple effects*.

candidates. The rest of the columns are summarized in the next section. From the remediation strategies of tools, we found three major causes of incompatibility issues. **(1) No reliable detection**: Although Steady and Dependabot support compatibility scores, their results were unreliable due to inaccuracy. **(2) Lack of global optimization**: Because vertices in DG were interconnected with each other, optimizations of them were not independent. Thus, it is impractical to optimize each vertex individually without a global perspective. **(3) Lack of support of handling *ripple effects***: The optimization was conducted based on the original DG without updating structures and call graphs. Then, the optimal solutions based on the new DG were changed so that the existing tools would return sub-optimal solutions.

C. Study of Users' Concerns with Remediation Suggestions

GitHub provides various automated SCA extensions to create PRs of security updates for dependencies, but these PRs are far from perfect, and thus sometimes rejected by users. To increase the acceptance rate of suggestions, we conducted a study to understand the concerns of users towards the remediation at GitHub by analyzing the reasons for rejected remediation suggestions and the accepted suggestions as a comparison.

Due to the lack of existing studies on Maven projects, the data set was collected by ourselves. First, we derived $9,527$ projects active in the last three years with 100+ starts at GitHub. Then, $5,356$ un-merged PRs created by bots were located and narrowed down to 306 PRs with human participation. Finally, we manually went through the comments in these PRs and summarized several reasons why PRs were unmerged.

- (91 cases, 29.74%) **Duplication**: The upgrades were superseded by other PRs, which were eventually merged.
- (82 cases, 26.80%) **Compilation/Test/CI failures and Dependency conflict (DC)**: The developers ran tests on the projects with upgraded dependencies, and incompatible issues occurred. Particularly, tests failed at dependency resolution, compilation, and test stages. For all PRs created by Dependabot in this category, compatibility scores were shown as *unknown*.
- (75 cases, 24.51%) **Incompatibility concerns**: The developers were concerned by incompatibility risks because either the upgrades had large spans, such as major upgrades, or they were known to be breaking. All compatibility scores were shown as *unknown* as well.
- (23 cases, 7.51%) **Internal errors**: Bots reported their internal errors in comments, so the users closed the PRs.
- (12 cases, 3.92%) **Unused dependencies**: The developers found the dependencies to be upgraded were not in use anymore, so the PRs were closed. The *bloated dependencies* were supposed to be ignored during the remediation.
- (9 cases, 2.94%) **Disobeying rules or absence of signed agreements**: The developers closed the PRs because the PRs failed to follow the rules of the repositories or sign the contributor agreements.
- (8 cases, 2.61%) **Unknown reasons**: The developers closed the PRs without explicitly mentioning the reasons.
- (6 cases, 1.96%) **Other**: There were various reasons: (1) Upstream projects demanded to keep the current version. (2) Java version was not compatible. (3) The PR introduced new CVEs. (4) Wrong user configuration. (5) A formatting issue.

From the result, excluding the duplicated PRs and unrelated reasons, such as internal errors, it is evident that the compilation/test failures and incompatibility concerns were the primary concerns of users (51.53%). The upgrades on unused dependencies could be avoided by the reachability analysis. The perspectives of concerns of users are demonstrated in Table I. *Dep conflict* refers to the support of the detection of possible dependency conflicts raised by Maven. The *ripple effects* denotes the support of dynamically handling the *ripple effects*. *Unused dependencies* means the support of detecting and ignoring unused dependencies.

Besides the reasons for rejected PRs, merged PRs were also studied as a comparison, but they usually failed to include the reasons for acceptance. Thus, we studied the distribution of their upgrades. Since the number of merged PRs is enormous, we studied the $556,257$ PRs merged in the last two years for Maven projects. The distribution was (1) *Major*: 11.91%; (2) *Minor*: 38.34%; (3) *Patch*: 48.55%; (4) *pre-release*: 0.89%; (5) No SemVer available: 0.31%. The result indicated that most merged PRs (87.79%) did not bump the versions to major upgrades, which followed the criteria of SemVer because non-major upgrades were supposed to maintain backward compatibility. Therefore, the remediation suggestions with fewer major upgrades are more likely to be accepted by users.

III. Methodology

A. Problem Formulation

By summarizing the users' concerns, we are able to define the objectives and constraints of the remediation. The primary objective is to minimize the total vulnerability risks:

$$min \quad F_{vul} = \sum_{m=1}^{M} \sum_{vul=1}^{Vul} \theta_{vul} f_{cvss}(vul) \tag{1}$$

where M is the number of libraries and Vul is the number of vulnerabilities of a vertex m. f_{cvss} is the Common Vulnerability Scoring System (CVSS) [15] weight. θ_v is the reachability coefficient for vulnerability v, particularly, θ_v is larger for reachable vulnerabilities, because the reachable vulnerabilities are possible to be exploited by attackers. However, in reality, not all vulnerabilities are open-source, which also increases the difficulty for attackers. Thus, the vulnerabilities with uncertain vulnerable classes or methods are classified as *unknown vulnerabilities* whose severity is ranked between the reachable and unreachable vulnerabilities. Since different vulnerabilities result in different risks, we use CVSS, a normalized score provided by NVD, to prioritize the vulnerabilities with higher risks during calculation.

The remediation is less likely to be accepted if it breaks the users' projects, according to the study in Section II-C. Thus, the pre-condition of successful remediation is the compatibility of version adjustments.

$$s.t. \quad c_{incom} = \sum_{m=1}^{M} \sum_{p=1}^{P} \theta_v * incom(v_p, v_m) = 0 \tag{2}$$

P is the number of parent vertices of v_m, while v_p is a parent vertex. c_{incom} is the total number of dependency relationships that cause incompatible issues. The incompatibility comprises two types of code-based breaking (semantic and syntactic breaking) and DC issues.

To achieve the global optimization and handle the *ripple effects* mentioned above, CORAL is supposed to optimize all connected vertices altogether in a dynamically adjusted DG. These goals bring three challenges: (1) Trade-off between the security and the compatibility during decision making. (2) The time complexity increase exponentially with the size of DG as $O(n) = \prod_{n=1}^{N}$ if all solutions are to be iterated over. (3) The *ripple effects* requires dynamically updated DG.

B. Overview

CORAL is implemented in four steps as illustrated in Fig. 1. (1) Generating DG and the call graph (CG) from the project object model (pom) file, a version control file of Maven, and class files of the project. (2) Partitioning the DG into subgraphs. (3) Optimizing the subgraphs regarding the vulnerability risks based on the pre-computed vulnerability mappings while ensuring compatibility. (4) Backtracking to parent vertices heuristically if the dead end is met. Then, the final remediation suggestions of version adjustment of all TPLs in the DG are returned.

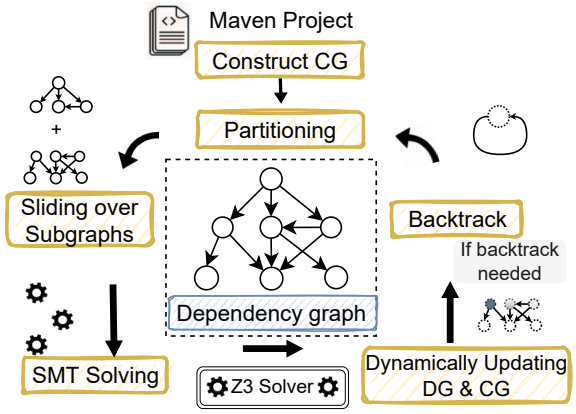

Fig. 1: Overview of CORAL

C. Constructing Dependency Graph and Call Graph

With pom files and class files, CORAL extracts the dependency tree by the Maven command and recovers the DG by completing the absent dependency relationships from a pre-computed dependency database. According to Maven documentation [16], as dependencies with *test* scope are not involved in the normal use of the projects, CORAL excludes dependencies with *test* scope from the DG. Specifically, DG is represented as $DG = Graph(V, E)$, where $V = \{e_i^x \mid i \in \{0, ..., N-1\}, x \in \{0, ..., L\}\}$ and $E = \{e_i \rightarrow e_j \mid i, j \in \{0, ..., N-1\}\}$. $\rightarrow$ denotes the direction of the calling edge, and x specifies the stack level w.r.t the DG.

The CG is constructed statically based on Soot [17] by the Spark algorithm [18] from the class files of the projects. The *main* methods are considered the entry points which serve as the start of the call graphs. If *main* methods are absent, we overestimate that it is possible to execute all methods implemented in the projects. Thus, all methods in users' projects are considered entry points. Since handling the *ripple effects* requires the dynamically updated CG to achieve real-time reachability analysis, the call edges in the CG are collected modularly. i.e. call edges are not extracted from a Uber jar [19] (root project with all dependencies) but from jars of each dependency separately and sequentially and then integrated into one graph originating from the root project. Particularly, for each dependency, the callers from the parent libraries serve as the entry points for child libraries. After the remediation, if the child libraries are suggested for other versions, the callees in them can be substituted accordingly to generate the real-time CG flexibly.

D. Partitioning Dependency Graph

Due to the high complexity of optimization over the entire DG, CORAL partitions the DG into subgraphs to reduce the size of the overall solution space. The partitioning comprises two steps, vertical partitioning, and horizontal partitioning. As illustrated in Fig. 2 (a), the vertical partitioning iteratively splits the DG into multiple partitions that are not connected

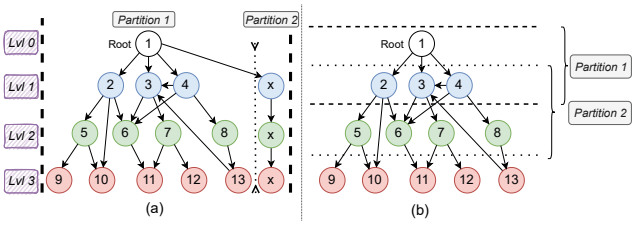

Fig. 2: Dependency Graph Vertical (a) and Horizontal (b) Partitioning

with each other by dependency edges except the direct relationships from the root project v_1 until all unconnected partitions are split. Since the direct dependencies do not depend on each other, optimizations on multiple partitions can be conducted independently and concurrently. For example, in Fig. 2 (a), *partition 1* and *partition 2* do not depend on each other. Hence, they can be partitioned to boost performance.

However, the vertical partitioning is not always sufficient, especially for the large partition at left in Fig. 2 (a). In this case, horizontal partitioning can further reduce the solution space. The subgraphs are partitioned by levels to preserve the semantics. According to [20], the semantics of a method decays along the calling chain, i.e. dependencies closer to the root matter more than those farther from the root in terms of the semantics or functioning they provide. For better notations, dependencies are labeled by tags called *level* to denote the smallest number of hops from the root. To better preserve the semantics of dependencies against the potential incompatibility, CORAL split DG and group vertices at level l and $l-1$ into subgraphs as in Fig. 2 (b). Then, because the closer dependencies preserve more semantics, CORAL starts the optimization from the root user projects in a top-down manner. Particularly, the lower-level dependencies should humor the upper ones in terms of the compatibility constraints as much as possible. Hence, CORAL attempts to optimize dependencies in two adjacent levels at a time and then moves the sliding window of a partition down to the next level with a newly updated CG. With the horizontal partition, the complexity can be reduced to $O(n) = \sum_{p_{hori}=1}^{P_{hori}} \sum_{p_{vert}=1}^{P_{vert}} \prod_{n=1}^{N_p}$. The side effect is that the potential better solution with lower vulnerability risks may be overlooked for dependency edges across multiple levels. To compensate for the loss, Section III-F introduces the backtracking mechanisms to avoid sub-optimal situations.

E. Optimizing Subgraphs

In this subsection, the detailed specification of the optimization on subgraphs based on Z3 SMT solver [21] is described.

*1) **Objectives and Constraints Definition**:* In each subgraph, CORAL conducts the optimization to minimize the vulnerability risks in the condition that the version changes are compatible. The vulnerability elimination follows the objective function in Equation (1). The basic vulnerability elimination strategy is to find versions with the fewest reachable and unknown vulnerabilities. Then, if more than one versions

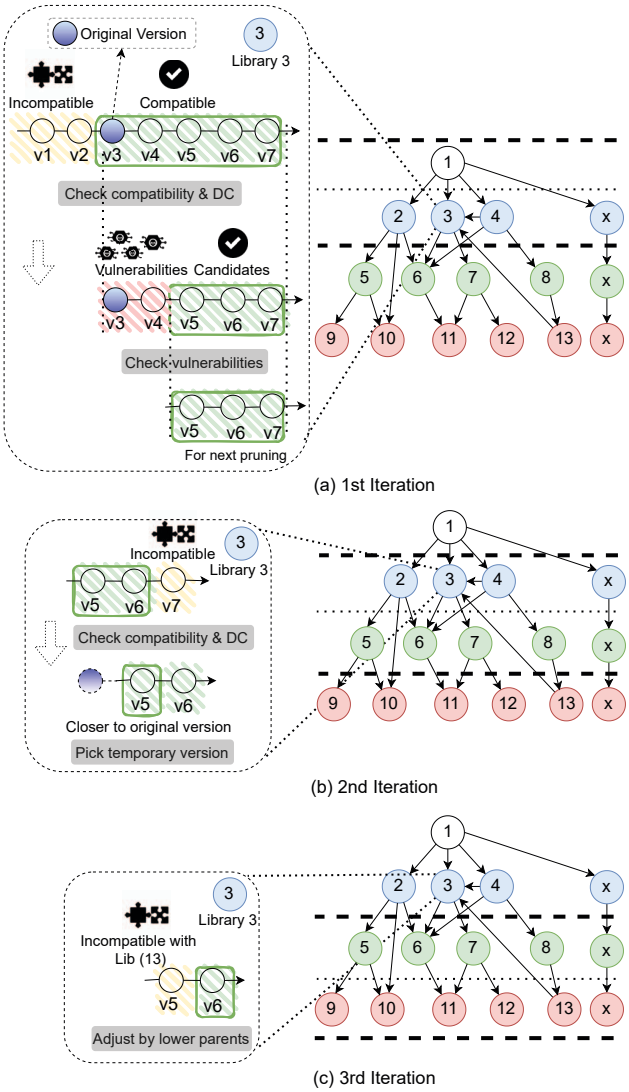

Fig. 3: Example of the Version Selection of a Dependency

satisfy these conditions and other constraints, the versions without unreachable vulnerabilities are preferred.

Theoretically, the compatibility constraint is supposed to be strict. However, not all types of incompatibility can be accurately detected. Generally, there are three major types that CORAL aims to resolve, namely, semantic breaking, syntactic breaking, and dependency conflicts, as discussed in Section II-C. Except for semantic breaking, the rest can be detected statically and efficiently. Thus, the detection of the rest is integrated into the optimization as constraints:

$$s.t. \quad c_{synb} = \sum_{m=1}^{M} \sigma * synb(P(x'_m), x'_m) = 0 \tag{3}$$

The $synb$, Syntactic Breaking, is calculated based on the reachability analysis and the API compatibility checkers. For each version pair of one library, the modified APIs that can

cause the failure of compilation are calculated by the three most widely used API compatibility checkers japi-compliance-checker [22], revapi [23], japicmp [24] based on the pair of jar files. Then, based on the reachability analysis, the called APIs of this library are obtained from CG. If any problematic APIs are called, the compilation would mostly fail, so CORAL would label this candidate version as breaking and discard it.

$$s.t. \quad c_{dc} = \sum_{m=1}^{M} dc(P(x'_m), x'_m) = 0 \tag{4}$$

The DC issues are calculated based on Maven version rules [25]. Like other package managers, version ranges define the allowed versions for dependencies. If two version ranges required by dependents do not overlap, Maven would report *Dependency Conflict* during version resolution before the compilation. A similar logic is implemented in CORAL to only select versions within the intersection of ranges defined by dependents. It is noteworthy that over 99% dependency version specifications are not determined with ranges, but single recommended versions instead, which means all versions are available regardless of compatibility. In this case, CORAL would include all versions as candidates for DC detection as Maven.

Since the semantic breaking is usually revealed by unit tests subject to limited coverage according to [26], it is hard to detect it statically and efficiently. Also, it is the leading cause of unit test failures [27], which is one of the main reasons why users reject remediation suggestions. Thus, CORAL relies on auxiliary information to infer the potential semantic breaking and minimize its probability by following the SemVer and Maven versioning guides. According to SemVer, the *Major* upgrades are allowed to break the original implementations. Hence, CORAL avoids using *Major* upgrades/downgrades as much as possible unless they are less vulnerable and satisfy the other compatibility criteria. Thus, besides the primary objective, we add a secondary objective, f_{major}, the number of dependencies that have *Major* upgrades/downgrades:

$$min \quad f_{major} = \sum_{n=1}^{N} f_{major,x_n}(x_n, x'_n) \tag{5}$$

$$\text{where } f_{major} = \begin{cases} 0 & \text{if } x_n \text{ to } x'_n \text{ is not major} \\ 1 & \text{if } x_n \text{ to } x'_n \text{ is major} \end{cases}$$

Although SemVer stipulates *Minor* should not include incompatible changes, researchers from [28] found that *Minor* upgrades are not as compatible as *Patch* upgrades, which generally introduce more breaking changes. Therefore, CORAL always prefers *Patch* upgrades rather than *Minor* if all other conditions stand. Another secondary objective function of f_{minor} is created to fulfill the purpose.

$$min \quad f_{minor} = \sum_{n=1}^{N} f_{minor,x_n}(x_n, x'_n) \tag{6}$$

$$\text{where } f_{minor} = \begin{cases} 0 & \text{if } x_n \text{ to } x'_n \text{ is not minor} \\ 1 & \text{if } x_n \text{ to } x'_n \text{ is minor} \end{cases}$$

Besides SemVer, Maven version control rules [25] also help identify potentially breaking versions. First, the pre-release versions, also known as development versions, such as *alpha, beta, SNAPSHOT* versions, are unstable and prone to breaking changes, which are selected at a lower priority than *Major* upgrades. Second, the larger version spans are usually more likely to induce incompatible changes. CORAL attempts to reduce the version span from the original version to the new version as much as possible. In terms of these two objectives, the functions of f_{dev} and f_{span} are formally given as:

$$min \quad f_{dev} = \sum_{n=1}^{N} f_{dev,x_n}(x_n, x'_n) \tag{7}$$

$$\text{where } f_{dev} = \begin{cases} 0 & \text{if } x'_n \text{ is not dev} \\ 1 & \text{if } x_n \text{ is not dev, } x'_n \text{ is dev} \end{cases}$$

$$min \quad f_{span} = \sum_{n=1}^{N} dist(x_n, x'_n) \tag{8}$$

where $dist(x, y)$ is the distance between x,y in sorted versions.

After solving with the SMT solver, each vertex in the subgraph is assigned with a selected version, and upgraded libraries in CG will be updated accordingly. However, the selected versions can be overthrown by the next optimization. Thus, all selectable candidate versions are saved and fed to the next optimization. For instance, in Fig. 3 (a), *Lib* 3 initially has 7 candidates and gets filtered to 3 by incompatibility and vulnerabilities. In the next iteration (b), *Lib* 3 has its candidates further filtered to 2 because of the incompatibility. Then, $v5$ is selected due to its smaller version span from the original version. However, in the third iteration (c), $v5$ is overthrown because it is not compatible with the parent library *Lib 13* at a lower level. Since the compilation and Maven resolution would fail regardless of the levels, the selected versions must follow the constraints in Equations (3) and (4). Therefore, $v5$ is discarded, and $v6$ with compatible changes is selected.

F. Backtracking

Although sequential partitions of DG reduce the complexity, they could lead to sub-optimal solutions and dead ends. To mitigate such issues, two types of backtracking mechanisms are implemented in CORAL, the hard and the soft backtracking.

*1) **Hard Backtracking**:* Hard backtracking is implemented to avoid dead ends. It happens during deciding the best version of a library where all versions disobey the constraints by potentially breaking the project. The backtrack targets are parent libraries of the current library. Since backtracking requires re-visiting the related vertices, the parent library at the lowest level is prioritized to reduce the efforts of re-visiting. And then, the higher ones are attempted if the lower parent triggers the backtrack again. During one backtrack, the selected version of the target parents is temporarily marked as incompatible, and other versions are attempted.

*2) **Soft Backtracking**:* Soft backtracking is used to avoid sub-optimal solutions. It is triggered when the version selected by the SMT solver is not the version with the lowest vulnerability risks in the version list, such as non-vulnerable versions. Like the hard backtrack, the soft backtrack prioritizes the parent libraries at lower levels. The different part

Algorithm 1: Algorithm of CORAL

Input: Dependency Graph $G\langle V, E\rangle$ (vertices V and edges E) with h levels, class files cf of the project
Output: Remediated $G'\langle V', E'\rangle$ with newly assigned versions

```
parts_v ← verticalPartition(G)
foreach part in parts_v do
    foreach i_th in h do
        part_h ← V_i + V_{i+1}
        cg ← CallGraph(part_h, cf)
        foreach v in V do
            parents ← parentsOf(v)
            foreach ver in versionsOf(v) do
                if ver has synb or DC then
                    cand.remove(ver)
                if sizeOf(cand) == 0 then
                    hardBacktrack
                    break
                vuls ← vulsOf(ver)
                foreach vul in vuls do
                    θ ← reachability(vul, cf)
                sort candidates by θ
            s ← SMTsolver(V_i, V_{i+1})
            if vuls(s)! = min(vuls)) then
                softBacktrack
                break
            cg ← updateBy(s)
            G ← updateBy(s)
            if hardBacktrack then
                p ← parent_lowest
                p.incompatible ← ver
                backtrack to p
            if softBacktrack then
                p ← parent_lowest
                runs ← saveVul(p)
                backtrack to p
                foreach r_th run in p.vers do
                    runs ← saveVul(p_r)
                s ← min(runs)
return G'⟨V', E'⟩
```

is that soft backtrack does not mark the parent's current version as incompatible but unpreferrable instead. It means if other versions are proven to be not as optimal as the unpreferrable version after the backtracking, the unpreferrable would still be selected. Thus, even if versions satisfy the constraints, they could be ignored by soft backtracking. During the soft backtracking, CORAL saves the overall vulnerabilities between the backtracked library and the target parent for future comparison. After the backtracking, CORAL compares the vulnerabilities of the current run with the ones saved previously and adopts the run with the fewest vulnerabilities to apply the versions to backtracked libraries accordingly. Note that to avoid an infinite loop, soft bakctracking would not be triggered again during one run of soft backtracking. Also, if the hard backtracking is triggered during soft backtracking, the current run would be discarded, and other versions would be attempted.

In conclusion, CORAL was designed to overcome the challenges of the high complexity of global optimization and *ripple effects*. The algorithm is presented in Algorithm 1. CORAL starts with vertical and horizontal partitions to split the DG into multiple parts. Then, the SMT solver is used to optimize the remediation results in each partition in a top-down manner. If any backtrack is triggered, CORAL backtracks to the previous vertices to avoid the sub-optimal solutions.

IV. EVALUATION

We aim to answer the following research questions:
RQ1: How is CORAL compared with other cutting-edge remediation tools regarding security and compatibility?
RQ2: How effectively does CORAL resolve the challenge of global optimization by subgraph partitioning?
RQ3: How many vulnerabilities CAN/CANNOT be fixed without breaking the projects in the Maven ecosystem?

A. Preparation

*1) **Data Collection**:* To build a data set of in-development Maven projects, we collected 301 most starred projects managed by Maven at GitHub on May 21st, 2022. We first selected Java projects with the most stars from GitHub and excluded non-Maven projects. Next, we manually modified the POM files of each project to apply the remediation suggestions from these tools. Considering the efforts of manual work, we filtered these projects with 1K+ stars. Finally, we got 301 selected projects. The demographics of the data set are illustrated in Fig. 4. It has the following features: (1) The code base size is non-trivial (average 22.19 kloc). (2) The range of sizes of dependency graphs is large (max 327, average 32.0). (3) The projects are affected by an adequate number of CVEs (average 27.6). (4) The projects are popular due to high star numbers.

To experiment with accurate vulnerability mappings, we periodically crawled CVE feeds from NVD [29] with a pipeline and pre-classified the language of CVEs by keyword matching. As the CVE descriptions are free-text [30], [31], it is impractical to directly extract version mappings from them. Hence, we manually triaged the mappings from reference links and associated Common Platform Enumerations (CPEs) [32]. So far on May 21st, 2022, we collected mappings for $1,759$ CVEs associated with Maven libraries. In this section, the evaluation needs the reachability analysis, which requires the vulnerable methods and classes associated with CVEs. Thus, we first identified 750 CVEs (42.64% of all Maven CVEs) from $2,326$ unique libraries used as dependencies in 301 projects. Then, vulnerable classes and methods of 300 CVEs were successfully identified and manually collected from the patches available at NVD links. The mappings and vulnerable methods of lib-vers and CVEs are publicly accessible on our website [10].

*2) **Tools and Environments Preparation**:* All tools used in the evaluation were tested with their latest versions in May 2022. Steady was tested with version $3.2.4$ with a built-in vulnerability database including 729 CVEs. The two commercial tools were evaluated in their publicly accessible production environments. CORAL was implemented with 6.9kloc in Python

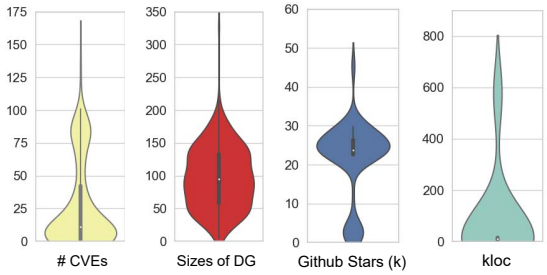

Fig. 4: Demographics of the Data Set of RQ1 and RQ2

3.8.2 and evaluated with java $7-13$ (depends on projects), Maven 3.8.2, and Ubuntu 18.04.6.

B. RQ1: Comparison with Other Remediation Tools

*1) **Evaluation Metrics**:* (1) *Vulnerability fixed*: The primary target of remediation is fixing vulnerabilities which are further classified in terms of reachability as remaining reachable CVEs: Vul_r, remaining unreachable/unknown CVEs: Vul_{ur}/Vul_{uk}, and total fix: Fix. (2) *Compilation*: $Fail_{comp}$. The projects with updated pom files were compiled by Maven to evaluate the correctness of Maven resolution and the compile-time compatibility. (3) *Unit test*: $Fail_{test}$. The affiliated unit tests were run against the remediated projects to evaluate the runtime compatibility. (4) *Supplementary metrics*: The number of upgraded/downgraded libraries, total version span, number of *Major* upgrades (#Major), and development upgrades (#Dev) were counted for reference.

*2) **Comparison Results**:* The evaluation was conducted based on the remediated projects (versions returned by remediation tools were adjusted in pom files), which along with the Maven logs, are available on our website. To emphasize the improvement gained from the version selection strategy, we added two baseline tools with naive strategies. Both baseline tools share the same partitioning and backtracking mechanisms as CORAL. *Baseline A* always prefers the latest versions of vulnerable libraries. It is used to demonstrate the result of a common practice which is upgrading vulnerable dependencies to the latest. *Baseline B* always prioritizes the versions with the fewest reachable and unknown vulnerabilities, even if it may break the projects. *Baseline B* gave an idea of how many non-trivial vulnerabilities could be fixed without being constrained by compatibility. The comparison results with remediation tools and baselines are provided in Table II. The analysis of each metric is supplied as follows:

- **Remaining Reachable Vuls**: Due to a limited number of vulnerable methods, only 17 CVEs could be identified as reachable in original projects. It is noteworthy that CORAL eliminated all reachable CVEs. Because Dependabot returned far fewer remediation suggestions than other tools, $16/17$ reachable CVEs remained reachable after remediation. As Steady's vulnerability database is limited, we re-evaluated Steady with the 729 CVEs in their database and enclosed the updated numbers in brackets. Within this scenario, Steady had fewer reachable CVEs than before, like other tools.
- **Remaining Unreachable and Unknown Vuls**: CORAL had much fewer unreachable vulnerabilities (reduce 87.56% of vulnerabilities) than other tools because though the unreachable vulnerabilities were considered harmless, CORAL attempted to remove them if the constraints allowed. Unknown vulnerabilities 583 still remained in the DG for three reasons: (1) $244, 41.87\%$. The versions with fewer vulnerabilities did not satisfy the constraints. (2) $149, 25.56\%$. All versions were vulnerable. (3) $101, 17.31\%$. The more secure versions with unreachable CVEs were *Major* upgrades with overly large version spans. Regarding baselines, *Baseline A* proved that upgrading to the latest fixed only an insignificant amount of vulnerabilities. *Baseline B* suggested that 338 (4.05%) more vulnerabilities could be fixed without considering compatibility.
- **Compilation Failures**: CORAL achieved 98.67% successful compilation rate due to detecting syntactic breaking and DC issues. The reasons for four failed cases were (1) Call graph generation failure: One of the libraries along the call chain had no call edges generated, which led to unreachable breaking methods. (2) Exception class not captured: The breaking exception class was not captured in the call graph and thus deemed unreachable. (3) Overriding not captured: The breaking methods of a class were extended and overridden in the new version, but the call graph did not reflect such overriding. For example, a project, *apollo-client* [33], had a failed compilation due to the incompatibility in its dependency, *snakeyaml*. The overriding of class, *BaseConstructor*, was not captured. (4) Ghost dependency: The breaking methods were used in an undefined dependency, so they were not captured as reachable methods. Because of the local optimization and unreliable or absent compatibility detection, the rest of the tools were subject to broken upgrades with failed compilation.
- **Unit Test Failures**: Since it is challenging to detect Semantic Breaking effectively, it is difficult to prevent Unit test failures. Thanks to the prioritization based on SemVer and Maven resolution rules, CORAL was able to achieve the fewest failures among these tools. Note that due to private dependencies, unfinished development, special requirements of running environments, etc., 88 unit tests in original projects already failed without remediation, which was excluded from the number of failures in the table.
- **Other Statistics**: It is evident that Com B had many more lib-ver pairs changed because it manipulated the direct dependencies to adjust the associated trees by changing the default versions of subsequent dependencies regardless of vulnerabilities, while other tools mostly focused on the vulnerable vertices. The same reason stood for the version span. Because CORAL, Steady, and Com B substantially changed the versions of transitive dependencies, their total version spans were larger than Dependabot's.

TABLE II: Comparison of CORAL among State-of-the-art Remediation Tools

Tool name	Avg DG Size	Vul_r	Vul_{ur}	Vul_{uk}	$Fixed_{CVE}$	$Fail_{comp}$	$Fail_{test}$	#Crashes	#Libs changed	Version span	#Dev	#Major
Original	33.99	17	5,363	2,954	0	0	0	0	0	0	0	0
CORAL	**36.27**	**0**	**553**	**583**	**7,198**(87%)	**4**	**15**	**0**	**2,556**	**70,464**	**3**	139
Dependabot	34.93	16	5,357	2,682	262 (3%)	20	31	1	602	17,024	0	44
Steady	44.17	11(4)	1,596(955)	1,457(515)	5,253(63%)	27	36	1	2,292	75,380	4	257
Com A	34.24	7	4,199	2,410	1,469 (18%)	51	61	7	1,398	24,679	0	245
Com B	35.61	3	1,040	1015	6,277(75%)	54	70	0	6,498	134,407	0	170
Baseline A	33.81	3	4,677	2,786	869	39	45	0	2,580	16,863	7	194
Baseline B	43.11	0	422	376	7,536	54	71	0	5,860	90,931	5	329
Baseline C	35.11	0	535	547	7,252	4	12	0	2,613	56,738	1	126

1) Vul_r: number of reachable CVEs. Vul_{ur}: number of unreachable CVEs. Vul_{uk}: number of *unknown* CVEs. $Fixed_{CVE}$: number of fixed CVEs. $Fail_{comp}$: number of projects with failed compilation. $Fail_{test}$: number of projects with failed tests. *Crashes*: number of projects that tools crashed and failed to return results. *Dev*: number of development version pairs. *Major*: number of *Major* version pairs

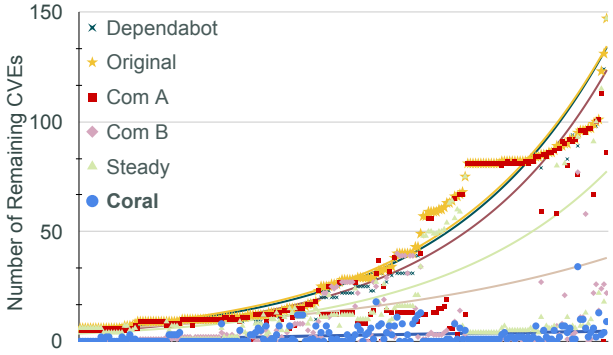

Fig. 5: Ascending Order by Numbers of CVEs of Original per Project

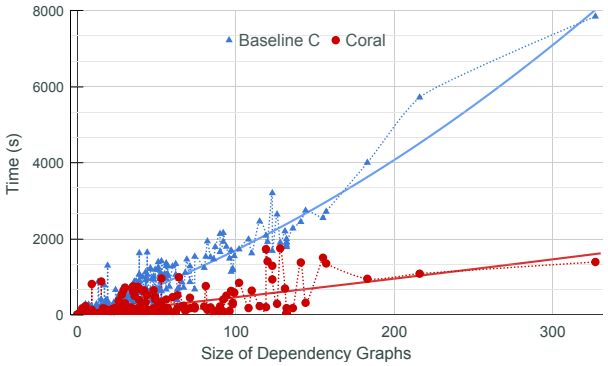

Fig. 6: Time Consumption of *Baseline C* and CORAL

To illustrate the distribution of remaining vulnerabilities over all projects, the remaining CVEs of all tools are presented in the scatter plot of Fig. 5. The x-axis is ordered by the number of CVEs of the original, which serves as the upper bound denoted by yellow stars. It is evident that CORAL has the overall fewest remaining CVEs at the bottom of the chart, denoted by blue dots.

Conclusions of RQ1: From the evaluation in Table II, CORAL fixed 87.56% of all CVEs with all reachable removed, including 911 more CVEs than the best of the rest tools. Meanwhile, CORAL achieved the 98.67% successful compilation rate and 92.96% successful unit test rate, which outperformed the rest of the tools. Compared with the two baseline tools, CORAL was proven to be effective at balancing the compatibility and security by breaking 106 (35.21%) fewer projects at the cost of 338 (4.05%) fewer vulnerabilities fixed.

C. RQ2:Effectiveness of Improvement on Global Optimization

The subgraph partitioning was implemented in CORAL to boost the performance towards the global optimization. To evaluate the effectiveness of partitioning, *Baseline C* was implemented in the same logic without two types of partitioning used by CORAL. The same data set was used to evaluate the existing metrics and time consumption. To measure the time, we respectively ran CORAL and *Baseline C* ten times against each project and calculated the average time as the final result. The result is presented in Fig. 6, which illustrates that the *Baseline C* generally tended to spend more time than CORAL for complete remediation. In the figure, the 301 projects are ordered by the size of DG. Each dot in the figure represents a single project. The tendency curves of both are fitted by the second-degree polynomials to avoid over-fitting.

To explain the fluctuations of the consumed time of CORAL, we manually analyzed the causes of the outliers. First, the 6 lower outliers were collected and analyzed. The cause of these cases was subgraphs partitioned were pretty small (1-5 deps), and the backtracking was not triggered. Second, for 18 higher outliers, there were four major causes:

- (9 cases) **Call graph generation failures**: The Call graph generation of the Soot script failed at some dependencies of DG, which took a long time to return. Usually, the failure of one version would persist with other versions of the same library, so the total time was elongated.
- (5 cases) **SMT solver took a long time**: For these projects, both *Baseline C* and CORAL spent a long time because the SMT Solver took a long time to finish. The direct reason for this cause was that the levels of DGs were limited, which means the DGs were more flattened than others. Thus, the

partitioning of CORAL based on levels could still include a substantial number of vertices in the SMT solver.

- (3 cases) **Multiple backtracking**: Another backtracking could be triggered during the current run of backtracking or after the current run fails. In these cases, 2 out of 3 cases had over three attempts of failed hard backtracking, and the rest triggered the hard backtracking multiple times during soft backtracking, which led to no improvement of vulnerability reduction for this soft backtracking.
- (1 case) **Jar downloading failure**: The CG generation and Syntactic breaking detection relied on the jar files of dependencies, CORAL failed to download from Maven Repository with time-out multiple times.

The observed metrics for *Baseline C* are presented in Table II. From the table, *Baseline C* has fixed 54 (0.75%) more vulnerabilities than CORAL, which implies the global optimization without partitioning has slightly improved the vulnerability fixing. Moreover, the number of projects with failed compilation stayed the same because CORAL handled the syntactic breaking and DC issues regardless of the partitioned subgraphs by backtracking.

Conclusions of RQ2: The comparison between *Baseline C* and CORAL substantiates that the partitioning mechanism could substantially reduce the time consumption without introducing the compilation failures at an acceptable cost of 0.75% fewer fixed vulnerabilities, especially for the large DG.

D. RQ3: How many fixable/unfixable CVEs in Maven

We target finding out how many vulnerabilities can be fixed without breaking the compilation and how many cannot in popular Maven projects. Since CORAL could efficiently exclude the solutions that broke the compilation with high precision (98.67%), we made an assumption that CVEs fixed by CORAL were fixable and CVEs not fixed by CORAL were unfixable.

*1) **Preparation of data**:* To conduct a large-scale study in the Maven ecosystem, we constructed a different data set from RQ1 and RQ2. Considering the balance between the representativeness and quality of the dataset, we first collected repositories with 100+ stars managed by Maven from GitHub to ensure the high quality of the dataset. Then, we compiled them and extracted dependency trees from them by the Maven command. If both steps succeeded, the dependency trees and class files were used as input for the remediation. Eventually, we randomly selected 2,000 out of 6,898 projects (average size 103.58) for the evaluation to make sure the dataset was representative. As for CVE mappings, the same mappings were used as RQ1 and RQ2. Since collecting vulnerable methods and classes is not as straightforward as version mappings, which requires much more effort for all CVEs, we decided not to conduct the reachability analysis of vulnerabilities in the experiment.

2) Results of RQ3: **Fixable:** The fixable CVEs are 10,109 (78.45%) as in Fig. 7. It is inferred that around 78% vulnerabilities could have been safely eliminated from the popular Maven projects without breaking the compilation to reduce the vulnerability risks of the ecosystem. We further calculated the distribution of the CVEs regarding the levels of the libraries and the types of upgrades that removed the CVEs. Although 78% seems to be a large number, the majority of them could not be fixed without domain knowledge or the aid of CORAL. According to Fig. 7, the proportion of vulnerabilities that could be fixed by adjusting direct dependencies was 11.71%, out of which 8.34% belonged to *Minor* and *Patch* upgrades.

As users can straightforwardly upgrade their direct dependencies to non-major versions to fix vulnerabilities on their own, we applied this naive method for the comparison with CORAL. The result showed that 25.71% of CVEs can be fixed by upgrading direct dependencies. Due to *ripple effects*, not only were 8.34% in direct dependencies fixed but more CVEs in transitive dependencies were also fixed. It is implied that without the aid of CORAL, the rest of the fixable vulnerabilities (52.74%) could not be fixed straightforwardly.

Unfixable: The number of unfixable CVEs was 2,777 (21.55%) as in Fig. 7, which could not be fixed by CORAL for three major reasons, the soft backtrack, all versions of a library were vulnerable, and the secure versions were incompatible. Reflected in Fig. 7, it is observed that the major reason was incompatibility which accounted for 60.10%. Note that the incompatibility did not count the Semantic Breaking because it could not be reliably detected. The minor reason, soft backtrack, refers to the vulnerabilities being left unfixed because the soft backtrack could not eradicate all CVEs, but minimized the overall vulnerabilities by ignoring some CVEs.

Although unfixable vulnerabilities cannot be easily removed without breaking the projects, some of them are removable at an acceptable cost. For example, if an API is deprecated and migrated to another, users only have to invoke the updated API and upgrade to the target version to fix the issue and vulnerabilities. Thus, if efforts to fix incompatibility are acceptable, more vulnerabilities can be fixed thoroughly with minimized efforts by quantifying efforts to fix the incompatibility.

Conclusions of RQ3: Through the experiments with the most starred projects on GitHub, we found that 78.45% of vulnerabilities could be fixed without breaking the compilation. However, without the aid of CORAL, only 25.71% could be fixed by upgrading the direct dependencies to non-major secure versions.

V. THREATS OF VALIDITY

The threat of CORAL is the static call graph reliance because only the static call graphs are not accurate enough to capture all possible call edges, which is one of the causes of the unit test failures in Section IV-B2. One typical example of inaccurate static call graphs is that static call graphs may miss invocations made by dynamic features, e.g., reflection. More-

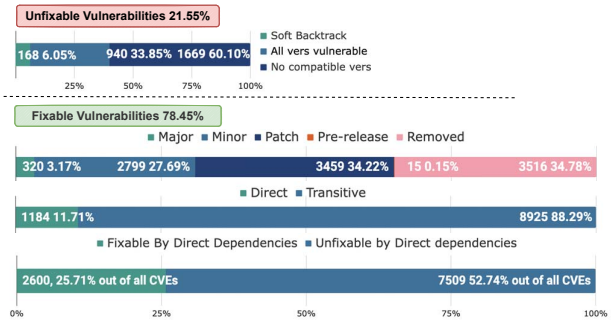

Fig. 7: Distributions of Fixable/Unfixable Vulnerabilities

over, The prioritization of vulnerabilities might overlook some reachable ones due to inaccurate static call graphs. However, to modularly and dynamically update the call graphs after each version adjustment, we could only generate static call graphs that are faster than dynamic ones. As it is impractical to run tests and generate dynamic call graphs thousands of times per project, we sacrificed accuracy for better performance.

VI. Related Work

A. SCA Tools and Strategies

SCA has been a popular research topic in recent years. Researchers have invested much effort to study and improve the two major procedures: component and vulnerability detection and vulnerability remediation.

*1) **SCA remediation**:* A limited number of research works [34]–[38] attempted to study and enhance the remediation strategy. Alfadel et al. [34] found for the Javascript projects at Github 34.58% of PRs created by Dependabot were not merged due to five reasons: (1) Duplication (2) Dependency conflict by peer requirements (3) Test failures (4) Internal errors (5) Disobeying rules/standards, which substantiate our findings in Section II-C. Steady [36], [37] has been developed for years to be a code-centric and usage-based SCA tool, which has been proved effective by Imtiaz et al. [39]. Soto et al. [38] found 22.6% of upgrades by Dependabot were recommended for bloated dependencies. 22.6% does not contradict our result 3.92% because 22.6% consists of all bloated dependencies, while ours were only those found and addressed. These works except for Steady mostly focused on the evaluation of remediation tools, which left a blank of remediation strategy enhancement filled by CORAL.

*2) **Component and vulnerability detection**:* Many researchers and practitioners [7], [8], [35], [39]–[48] have studied the component and vulnerability detection. Imtiaz et al. [39] studied 9 commercial SCA tools and found the reported vulnerabilities vary substantially, which revealed that the vulnerability database was the key differentiator. Dann et al. [40] reviewed six commercial and academic SCA tools regarding their ability to handle the dependency modification types. By testing 7k+ Java projects, they found the re-bundle modification in Maven dependencies was not supported by any tools. *Vuln4real* [35], [42] was proposed to exclude the false alarms of vulnerabilities by identifying the vulnerabilities in lagging, development-only, and unreachable dependencies, which significantly reduces false alerts.

B. Study of Open-source Software Ecosystem

Apart from SCA techniques, researchers [49]–[59] have studied the open-source software (OSS) and associated vulnerabilities in the OSS ecosystem, conclusions of which can be used to guide the designs of SCA tools. Decan et al. [49] studied NPM and Rubygems package managers and found that 33% and 40% of vulnerabilities respectively had their fixes within the same major release. Plate et al. [58] proposed new metrics to determine the criticality of vulnerabilities regardless of the types and languages of vulnerabilities, which helps with the automated impact assessment of new vulnerabilities. Imtiaz et al. [50] studied the characteristics of security fixes at 6 major package managers, namely, the semantic versions, release notes, and the time lag between fixes and releases, and offered 4 recommendations for the better practice of security releases. Ponta et al. [51] manually collected 625 publicly disclosed vulnerabilities for Java projects, which was also used in the Section IV as the Steady data set at the latest version.

VII. Conclusion

We proposed CORAL to provide remediation without breaking compatibility. The evaluation demonstrated that CORAL outperformed other tools by fixing 87.56% of vulnerabilities and achieving 98.67% successful compilation rate and 92.96% successful unit test rate. In the ablation study, the partitioning of DG and trade-off between security and compatibility had been proved effective. Furthermore, we found that 78.45% of vulnerabilities in popular Maven projects could be fixed without breaking the compilation.

Data Availability

The data sets of the studies and evaluations can be publicly accessed at *https://sites.google.com/view/icse23remediation*.

Acknowledgements

This research is partially supported by the National Research Foundation Singapore and DSO National Laboratories under the AI Singapore Programme (AISG Award No: AISG2-RP-2020-019), the NRF Investigatorship NRF-NRFI06-2020-0001, the National Research Foundation through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) project under the National Cybersecurity R&D (NCR) Grant award no. NRF2018NCR-NSOE003-0001, the Ministry of Education, Singapore under its Academic Research Fund Tier 2 (MOE-T2EP20120-0004) and Tier 3 (MOET32020-0004). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

REFERENCES

[1] "Log4j Vulnerability," https://nvd.nist.gov/vuln/detail/CVE-2021-44832, 2021.

[2] "Log4j Remote Code Execution," https://www.netskope.com/blog/cve-2021-44832-new-vulnerability-found-in-apache-log4j, 2021.

[3] "Software Composition Analysis," https://snyk.io/series/open-source-security/software-composition-analysis-sca/, 2022.

[4] "Eclipse Steady," https://projects.eclipse.org/proposals/eclipse-steady, 2022.

[5] "Dependabot," https://github.com/dependabot, 2022.

[6] "Snyk," https://snyk.io/, 2022.

[7] "White Source," https://www.whitesourcesoftware.com/, 2022.

[8] "OWASP Dependency Check," https://owasp.org/www-project-dependency-check/, 2022.

[9] "Scantist," https://scantist.com/, 2022.

[10] "Data set," https://sites.google.com/view/icse23remediation, 2022.

[11] "Maven," https://https://maven.apache.org/, 2022.

[12] "Commons-lang," https://github.com/apache/commons-lang, 2022.

[13] "Dependabot upgrade resulted in build failure," https://github.com/apache/commons-lang/pull/826, 2021.

[14] "Wala," https://github.com/wala/WALA, 2022.

[15] "Common Vulnerability Scoring System," https://nvd.nist.gov/vuln-metrics/cvss, 2022.

[16] "Maven Scope," https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html#Dependency_Scope, 2022.

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

[18] "Soot Spark Call Graph," https://soot-build.cs.uni-paderborn.de/public/origin/develop/soot/soot-develop/options/soot_options.htm#phase_5_2, 2021.

[19] "Uber-jar," https://imagej.net/develop/uber-jars, 2022.

[20] A. Schroter, A. Schröter, N. Bettenburg, and R. Premraj, "Do stack traces help developers fix bugs?" in *2010 7th IEEE Working Conference on Mining Software Repositories (MSR 2010)*. IEEE, 2010, pp. 118–121.

[21] "Z3 Solver," https://github.com/Z3Prover/z3, 2022.

[22] "japi-compliance-checker," https://lvc.github.io/japi-compliance-checker/, 2019.

[23] "revapi," https://revapi.org/revapi-site/main/index.html, 2021.

[24] "japicmp," https://siom79.github.io/japicmp/, 2022.

[25] "Maven Versions," https://maven.apache.org/pom.html#Version_Order_Specificationl, 2022.

[26] L. Chen, F. Hassan, X. Wang, and L. Zhang, "Taming behavioral backward incompatibilities via cross-project testing and analysis," in *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, 2020, pp. 112–124.

[27] S. Mostafa, R. Rodriguez, and X. Wang, "Experience paper: a study on behavioral backward incompatibilities of Java software libraries," in *Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2017, pp. 215–225.

[28] L. Xavier, A. Brito, A. Hora, and M. T. Valente, "Historical and impact analysis of api breaking changes: A large-scale study," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2017, pp. 138–147.

[29] "National vulnerability database," https://nvd.nist.gov/, 2022.

[30] H. Guo, S. Chen, Z. Xing, X. Li, Y. Bai, and J. Sun, "Detecting and augmenting missing key aspects in vulnerability descriptions," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 31, no. 3, pp. 1–27, 2022.

[31] H. Guo, Z. Xing, S. Chen, X. Li, Y. Bai, and H. Zhang, "Key aspects augmentation of vulnerability description based on multiple security databases," in *2021 IEEE 45th Annual Computers, Software, and Applications Conference (COMPSAC)*. IEEE, 2021, pp. 1020–1025.

[32] "Common Platform Enumeration," https://nvd.nist.gov/Products/CPE, 2022.

[33] "Apollo project," https://github.com/ApolloAuto/apollo, 2022.

[34] M. Alfadel, D. E. Costa, E. Shihab, and M. Mkhallalati, "On the use of dependabot security pull requests," in *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR)*. IEEE, 2021, pp. 254–265.

[35] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vuln4real: A methodology for counting actually vulnerable dependencies," *IEEE Transactions on Software Engineering*, 2020.

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

[37] Ponta, Serena Elisa and Plate, Henrik and Sabetta, Antonino, "Detection, assessment and mitigation of vulnerabilities in open source dependencies," *Empirical Software Engineering*, vol. 25, no. 5, pp. 3175–3215, 2020.

[38] C. Soto-Valero, T. Durieux, and B. Baudry, "A longitudinal analysis of bloated Java dependencies," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1021–1031.

[39] N. Imtiaz, S. Thorn, and L. Williams, "A comparative study of vulnerability reporting by software composition analysis tools," in *Proceedings of the 15th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2021, pp. 1–11.

[40] A. Dann, H. Plate, B. Hermann, S. E. Ponta, and E. Bodden, "Identifying challenges for OSS vulnerability scanners-a study & test suite," *IEEE Transactions on Software Engineering*, 2021.

[41] M. Alfadel, D. E. Costa, and E. Shihab, "Empirical analysis of security vulnerabilities in Python packages," in *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2021, pp. 446–457.

[42] I. Pashchenko, H. Plate, S. E. Ponta, A. Sabetta, and F. Massacci, "Vulnerable open source dependencies: Counting those that matter," in *Proceedings of the 12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, 2018, pp. 1–10.

[43] "Blackduck," https://www.synopsys.com/software-integrity/security-testing/software-composition-analysis.html, 2022.

[44] "Sourceclear," https://www.sourceclear.com, 2022.

[45] "Sonarqube," https://www.sonarqube.org/, 2022.

[46] L. Zhang, C. Liu, Z. Xu, S. Chen, L. Fan, B. Chen, and Y. Liu, "Has my release disobeyed semantic versioning? static detection based on semantic differencing," 2022. [Online]. Available: https://arxiv.org/abs/2209.00393

[47] X. Zhan, L. Fan, S. Chen, F. We, T. Liu, X. Luo, and Y. Liu, "Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in Android applications," in *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 2021, pp. 1695–1707.

[48] X. Zhan, L. Fan, T. Liu, S. Chen, L. Li, H. Wang, Y. Xu, X. Luo, and Y. Liu, "Automated third-party library detection for Android applications: Are we there yet?" in *2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2020, pp. 919–930.

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

[50] N. Imtiaz, A. Khanom, and L. Williams, "Open or sneaky? fast or slow? light or heavy?: Investigating security releases of open source packages," *IEEE Transactions on Software Engineering*, 2022.

[51] S. E. Ponta, H. Plate, A. Sabetta, M. Bezzi, and C. Dangremont, "A manually-curated dataset of fixes to vulnerabilities of open-source software," in *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*. IEEE, 2019, pp. 383–387.

[52] H. Perl, S. Dechand, M. Smith, D. Arp, F. Yamaguchi, K. Rieck, S. Fahl, and Y. Acar, "Vccfinder: Finding potential vulnerabilities in open-source projects to assist code audits," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 426–437.

[53] S. S. Alqahtani, E. E. Eghan, and J. Rilling, "Tracing known security vulnerabilities in software repositories–a semantic web enabled modeling approach," *Science of Computer Programming*, vol. 121, pp. 153–175, 2016.

[54] C. Liu, S. Chen, L. Fan, B. Chen, Y. Liu, and X. Peng, "Demystifying the vulnerability propagation and its evolution via dependency trees in the NPM ecosystem," 2022.

[55] J. Hejderup, "In dependencies we trust: How vulnerable are dependencies in software modules?" 2015.

[56] Z. Li, D. Zou, S. Xu, H. Jin, H. Qi, and J. Hu, "Vulpecker: an automated vulnerability detection system based on code similarity analysis," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*, 2016, pp. 201–213.

[57] K. A. Farris, A. Shah, G. Cybenko, R. Ganesan, and S. Jajodia, "Vulcon: A system for vulnerability prioritization, mitigation, and management," *ACM Transactions on Privacy and Security (TOPS)*, vol. 21, no. 4, pp. 1–28, 2018.

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

[59] W. Tang, Z. Xu, C. Liu, J. Wu, S. Yang, Y. Li, P. Luo, and Y. Liu, "Towards understanding third-party library dependency in C/C++ ecosystem," in *37th IEEE/ACM International Conference on Automated Software Engineering*, 2022.