UPCY: Safely Updating Outdated Dependencies

Andreas Dann
CodeShield GmbH
Paderborn, Germany
andreas.dann@uni-paderborn.de

Ben Hermann
Technical University Dortmund
Dortmund, Germany
ben.hermann@cs.tu-dortmund.de

Eric Bodden
Heinz Nixdorf Institute &
Fraunhofer IEM
Paderborn, Germany
eric.bodden@uni-paderborn.de

***Abstract*—Recent research has shown that developers hesitate to update dependencies and mistrust automated approaches such as Dependabot, since they are afraid of introducing incompatibilities that break their project. In fact, such approaches only suggest naïve updates for a single outdated library but do not ensure compatibility with other dependent libraries in the project. To alleviate this situation and support developers in finding updates with minimal incompatibilities, we present UPCY. UPCY applies the min-(s,t)-cut algorithm and leverages a graph database of Maven Central to identify a list of valid update steps to update a dependency to a target version while minimizing incompatibilities with other libraries. By executing 29,698 updates in 380 projects, we compare the effectiveness of UPCY with the naïve updates applied by state-of-the-art tools. We find that in 41.1% of the cases where the naïve approach fails UPCY generates updates with fewer incompatibilities, and even 70.1% of the generated updates have zero incompatibilities.**

***Index Terms*—Semantic versioning, Library updates, Package management, Dependency management, Software maintenance**

I. INTRODUCTION

Building software on top of open-source libraries and frameworks is an established practice in commercial and open-source Java projects to save costs and improve quality [1]. However, the recent Log4Shell vulnerability has shown that developers need to update outdated open-source libraries quickly when a new vulnerability has been discovered. While package managers, like Gradle and Maven, ease the updating process by automatically integrating libraries as dependencies and dependencies of those (so-called transitive), developers still need to find a set of update steps manually to update the library to a non-vulnerable version while also avoiding to break the project. For a library that developers directly integrated, finding an update is straightforward. However, developers often need to update other libraries as well to avoid incompatibilities. For a transitive library, it is even non-trivial to figure out how to update it to a target version, as developers have no direct control of its version. Consequently, studies [2], [3], [4], [5], [6], [7], [8], [9], [10] show that most developers hesitate to update dependencies since they are afraid that an update may introduce unexpected regressions or unintended side-effects. As a result, software systems remain vulnerable to attacks through known vulnerabilities for extended periods.

State-of-the-art tools like Greenkeeper [11], Dependabot [12], and Renovate [13] create pull requests for updating a library and check if the update breaks the project by executing the compile and test command. Crucially, since these tools rely on a project's dependency test coverage, which is oftentimes too low, they can only detect conflicting updates and breaking changes to a small extent [14], [3], [6]. Especially, calls between (transitive) libraries are only rarely covered by project's tests; a recent study by Hejderup et al. [3] reports a coverage of less than 21%.

Moreover, these tools only suggest updates for the single outdated library but do not check if other libraries in the project that depend on the outdated library must be updated as well. In fact, a safe backward compatibility update must satisfy multiple types of compatibility [1], [15], [3], [7]: if the project uses the library directly, the update must be source code compatible such that the project continues to compile; if the dependency is transitive the update must be binary compatible such that other libraries link with the new version; if the library belongs to a framework, all libraries of that framework must be updated consistently to ensure consistent runtime behavior. Discovering, debugging, and resolving these incompatibilities manually is cumbersome, and thus the main reason that discourages developers from updating [10], [6]. Especially, resolving incompatibilities between (transitive) libraries is challenging as current tools only check for updates for each library in isolation, while a project assembles more than 36 dependencies on average that partially depend on each other [16].

Existing work [3], [15], [1] provides strategies to assess breaking changes between two versions of a library; either source code, binary, or semantic-breaking compatibility issues, but does not check relations between dependencies nor supports developers for selecting updates that minimize the number of incompatibilities to other libraries.

To remediate this situation, we present the graph based-approach UPCY to support developers in eliminating vulnerable or outdated dependencies (especially transitive ones) from projects' dependencies. UPCY finds a list of update steps that updates a Maven dependency to a target version while minimizing the number of incompatibilities respecting its use, conflicting and framework dependencies, as well as source code and binary incompatibilities. To this end, we built a Neo4j graph database of the Maven Central repository representing all artifacts, and their direct and transitive dependencies. UPCY then maps the conditions that an update has to satisfy to graph constraints for Cypher (Neo4j's query language) and uses a min-(s,t)-cut algorithm to identify update steps with minimal incompatibilities.

We evaluate the approach on 1,325 well-tested, open-source

Java Projects sampled from GitHub by Hejderup et al. [3]. Our evaluation shows that UPCY successfully finds updates with fewer incompatibilities in 41.1% of the cases in which state-of-the-art approaches fail, importantly, 70.1% of these updates have zero incompatibilities. UPCY's computed suggestions require the update of two libraries on average to achieve safe backward compatible updates, indicating that compatibility can be maintained with reasonable effort.

To summarize, this work makes the following contributions:

- a comprehensive definition of the conditions a safe backward compatible update has to satisfy (Section III),
- UPCY - an approach for finding safe updates using min-(s,t)-cuts on a unified dependency graph, and their mapping to Cypher to query Neo4j (Section IV)
- an evaluation of the approach on 1,325 well-tested, open-source Java projects (Section V).

II. Dependency Management in the Maven Ecosystem

Java package managers, like Maven and Gradle, provide tooling to ease the distribution, maintenance, and inclusion of external third-party libraries from public open-source repositories like Maven Central. To include a specific library, developers specify the library's unique identifier in the form of the triple: group, artifact, and version (GAV) in a project's configuration file (pom.xml). The package manager automatically downloads, includes, and configures the library as a dependency of the project.

Based on the declared dependencies in the configuration file, the package manager builds the project's dependency graph, specifying the dependencies the project includes and their relations. Figure 1 shows an example dependency tree. In this graph, nodes represent libraries - the root node represents the project itself, and directed edges connect to dependent libraries.

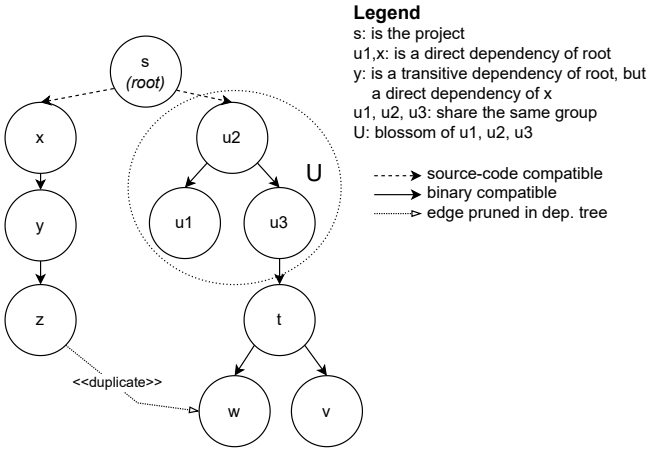

Fig. 1. Maven Dependency Tree

A dependency is called a *direct* dependency of node n if the dependency and n are connected through a path of length one. Dependencies connected through a longer path are called a *transitive* dependency of node n.

Although often visualized as a tree, strictly mathematically speaking, a project's dependency relations form a graph: a single dependency node can have multiple predecessors since multiple dependencies may depend on the same library, so-called duplicates. For instance, the dependency t and z both depend on dependency w in Figure 1. Similarly, multiple dependencies may depend on the same dependency in different versions, causing a so-called (version) conflict.

Java only supports a flat, linear classpath. Thus, a package manager transforms the dependency graph into a directed, rooted dependency tree. To create the dependency tree and to resolve ambiguous relations like circular dependencies, duplicates, or conflicting versions, the package manager Maven automatically picks the dependency that has the shortest path from the root node, thereby shadowing all other versions of that dependency [17], e.g., in Figure 1 the edge marked as *<<duplicate>>* would be pruned in the dependency tree.

III. Backward Compatible Updates

When developers aim to update a dependency, they have to consider any update step carefully since their project may unexpectedly suffer from regression-inducing changes, such as bugs or semantic changes that break API contracts [3].

A. Dependency Graph Updates

Depending on the position of a library l in the dependency tree, developers have to choose between multiple update options to maintain compatibility with other libraries [18].

- if l is a direct dependency of the project, it can be simply updated in the project's pom.xml, e.g., $u1$ in Figure 1
- if l is a transitive dependency, developers cannot simply increase their version as this would mean editing the source code of a direct dependency over which they usually have no control (or would require a fork). But they can
  - transform the updated version of l to a direct dependency of the project. Due to Maven's shortest-path resolution mechanisms, the direct dependency then shadows all other (transitive) instances of this dependency,
  - or check if any predecessor $p \in pred(l)$ in the dependency graph, lying on the path from project node to l, can be updated to a newer version that itself depends on a newer version of l, e.g., x of y in Figure 1. If such a predecessor p exists, developers can transform p to a direct dependency, thereby implicitly again shadowing the existing, former version of l.

In all cases, developers have to assess the effect of the library update on the project's dependency graph, since every update can introduce new (direct and transitive) dependencies that may lead to conflicts or shadowing of existing libraries. For instance, consider the duplicate dependency w in Figure 1. If the dependency t is updated, and its new version t' depends on a new version w' then w' will shadow the dependency w of z – whether desired or not.

With an increasing number of dependencies, duplicates, and conflicts, the complexity to maintain compatibility between all libraries in the dependency graph increases. Developers

must ensure that all nodes in the dependency graph that use the updated library continue to compile, link, and execute successfully [15]. A study by Wang et al. [1] showed that this is difficult for developers to diagnose in practice, leading to runtime exceptions and unexpected program behavior.

B. Source and Binary Compatibility

To ensure that an update is safe, the application programming interface (API) of the updated library must be compatible with the original one [15]. In particular, the API types, methods, and fields must not be subject to changes that prevent compilation, linking, or execution of formerly valid client code.

In Java, but also other compiled languages, it makes sense to distinguish between source and binary compatibility issues [19], [15]. Source compatibility issues result in compilation errors when the project is re-compiled against the updated library. Binary compatibility issues result in conflicts in the application binary interface (ABI). Binary compatibility errors lead to failures during linking or invocation [20].

Within a dependency graph, a library must fulfill exactly one of the two compatibility types for each incoming edge, depending on the nature of the dependency that the edge represents, shown in Figure 1. To be able to re-compile the project successfully, source code compatibility must be fulfilled for all libraries (transitive and direct) whose APIs are *directly* invoked in the project's source code. Source code *in*compatible changes are, for instance, removing public methods, types, or adding checked exceptions to an API method, as these exceptions must be handled by client code.

Binary compatibility must be fulfilled if a library is used by other dependencies that cannot be re-compiled. All dependencies in the dependency graph must continue to link successfully to the bytecode classes in the updated library. Examples of ABI *in*compatible changes are removing API types, methods, fields, or changes in the method's signature.

Note that binary compatibility does not imply source compatibility nor vice versa, although some compatibility issues affect both, e.g., the removal of API types [15].

A study from Dietrich et al. [15] shows that source and binary compatibility are regularly violated between different versions of a library. Although tools like SigTest [21] check if the API of a newer version is source code and binary backward compatible, they do not support reasoning about the behavior of several, intertwined libraries in the dependency graph. As shown in a study by Bogart et al. [14], even approaches like *Semantic Versioning* fail to indicate compatibility, as library maintainers release new changes based on their self-interpretation of backward compatibility.

C. Semantic Compatibility

Checking libraries for source and binary inconsistencies is a necessary precondition for a safe update but is insufficient: *semantic* compatibility is also required. Libraries with semantic compatibility issues are source and binary compatible, yet introduce *in*compatible runtime behavior that invalidates formerly valid assumptions made in the client code, e.g., a method formerly accepted null values as arguments while the updated version throws a NullpointerException.

A sound and precise detection of semantic compatibility issues is undecidable, however, several approaches exist that use or synthesize test cases to detect semantic compatibility issues w.r.t. the way the project uses a library [3], [1].

D. Blossom Compatibility

A study by Pashchenko et al. [22], found that projects commonly include dependencies that share the same group, indicating that they belong to the same framework. These dependencies are highly interconnected with each other and usually require the same version to run correctly. If a single library is updated to a new version, all other libraries of the framework must be updated. We refer to libraries with the same group in the dependency graph as dependency blossoms: they can be merged into a single dependency node like regular blossoms in graphs, see Figure 1. To achieve safe backward compatible updates, developers have to consider such blossoms as typically only those are guaranteed to work together by the framework authors [22].

A safe backward compatible update does not require complete source code, binary, and semantic compatibility. Instead, only those API types and methods that are actually invoked during the project's runtime must be compatible.

IV. UpCy: Identify Safe Updates

The available options for updating a library depend on its position in the dependency graph: (1) a *direct dependency* can only be explicitly updated, (2) a *transitive dependency* can either be transformed to a direct dependency, or any of its predecessors can be transformed to a direct dependency and updated if that predecessor depends on the new version. To reduce the risks of introducing regressions and to keep the effort for adapting the project low, one has to choose an update option that leads to the least amount of source, binary, semantic, dependency tree, and blossom incompatibilities.

To find compatible updates automatically, we created UpCy. Given a Maven project, a dependency, and its target version, UpCy finds a list of steps to update that library with minimum incompatibilities. To do so, UpCy computes a min-(s,t)-cut on the dependency graph with minimal violated compatibilities and runs Neo4j queries against the full Maven Central dependency graph to identify compatible updates of the dependencies that lie on the cut. The output of UpCy is a set of libraries that developers should add as direct dependencies to update the library to the target version, as well as the incompatibilities the update steps introduce (if any).

UpCy's approach is shown in Algorithm 1. For a given Maven project, library *libToUpdate*, and target version *targetVersion*, UpCy computes a set of list of update steps *listUpdateSteps*. The computed *listUpdateSteps* are dependencies developers should add as direct dependencies to their project to update the library to the target version with minimal incompatibilities.

Algorithm 1 UpCy - Identifying Safe Backward Updates

Input: libToUpdate, targetVersion
Output: listUpdateSteps

```
    /* build unified dep. graph */
 1: depGraph ← buildDependencyGraph()
 2: callGraph ← buildCallGraphSoot()
 3: unifiedDepGraph ← unify(callGraph, depGraph)
    /* naive Update */
 4: updateGraph ← queryNeo4j(libToUpdate, targetVersion)
 5: incmpts ← computeIncompatibilities(unifedDepGraph, update-
    Graph)
 6: if incmpts = ∅ then
    /* the root node of the updateGraph is the updated library */
 7:   return (getRootNodes(updateGraph), incmpts)
 8: end if
    /* update based on min-(s,t)-cut */
 9: minCuts ← computeMinCuts(unifedDepGraph, libToUpdate)
10: for minCut ∈ minCuts do
11:   query ← mapToCypher(minCut, unifedDepGraph)
12:   updateGraph ← queryNeo4j(query)
13:   incmpts ← computeIncompatibilities(unifedDepGraph, update-
    Graph)
    /* root nodes of the updateGraph are the nodes to add as direct deps */
14:   if incmpts = ∅ then
15:     return (getRootNodes(updateGraph), incmpts)
16:   else
17:     listUpdateSteps ∪ (getRootNodes(updateGraph), incmpts)
18:   end if
19: end for
20: return listUpdateSteps
```

First, UPCY builds the project's unified dependency graph to assess which API calls the project and libraries invoke. The unified dependency graph maps the sources and targets of API calls to the libraries in the dependency graph. To create the unified dependency graph, UPCY combines the project's dependency graph with the project's call graph that is created by passing the project's classes and dependencies as an input to the static analysis framework Soot [23] (lines 1-3).

Second, UPCY checks if a naïve update, which is just increasing the version number of the library, leads to incompatibilities (lines 4-8).

Third, UPCY tries to find an update with fewer incompatibilities than the naïve update using the min-(s,t)-cut algorithm [24] if the naïve update yields incompatibilities (line 9-19). To do so, UPCY computes all compatibilities the update has to fulfill, maps them to Neo4j's query language Cypher, and runs the query against our graph database of Maven Central (lines 11-12). The query returns the compatible libraries and their direct and transitive dependencies in the form of an *updateGraph* (line 12), with the dependencies to add to the project.

Fourth, based on the *updateGraph* and *unifiedDepGraph*, UPCY assesses the changes in the project's dependency tree and computes which API calls suffer from incompatibilities given the new libraries in the *updateGraph* (line 5, 13). We next detail further the various steps involved.

A. Building the Dependency Graph

To build the project's dependency graph, UPCY uses the Maven dependency graph plugin [25]. The plugin applies Maven dependency resolution mechanisms and computes all (transitive) dependencies the project includes. Figure 2 shows an example a project dependency graph with project s, blossom u, and the library to update t.

The plugin allows the construction of the complete dependency graph, including duplicate and conflicting dependencies, which are not part of the actual dependency tree. Duplicate dependencies occur when the project or dependencies defines the same dependency, e.g., w is a duplicate dependency because it is included by z and t. Conflicting dependencies occur when the project or dependencies defines the same dependency but with different versions. As described in the Section II, Maven will pick the dependency closest to the project (the shortest path in the dependency graph), thereby shadowing all other versions.

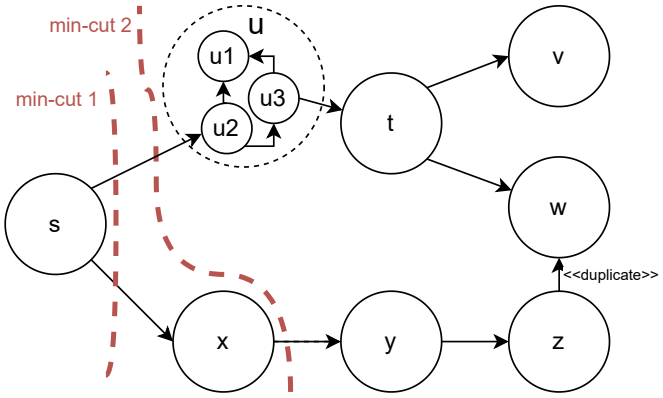

Fig. 2. Exemplary min-(s,t)-cuts computed by UPCY for updating t

B. Identifying Library API Usage

We refer to the use of libraries as using API types, methods, and fields from externally developed vendors. In particular, we focus on the use of API methods in libraries as they are among the most common forms of library usage. Thereby, we do not distinguish between the use of direct or transitive libraries but consider both equally. To identify the use of (transitive) dependencies, UPCY statically constructs the call graph representing the call paths between the project, its dependencies, and the calls between the dependencies themselves, excluding calls to the project or the Java Standard Library. To construct the call graph, UPCY uses the class hierarchy analysis of the static analysis framework Soot [23] with all project's classes as entry points, similar to Ponta et al. [22].

To create the unified dependency graph, UPCY maps the sources and targets of the found call edges to the libraries in the dependency graph. Each edge in the resulting graph represents a set of API calls that must be *binary* or *source*, and *semantically* compatible.

C. Graph Database of Maven Central

To explore the options for updating a specific library, all versions of that library must be known to UPCY. Furthermore, to evaluate which libraries the updated project will actually include and which will be shadowed, UPCY needs to simulate the impact of the update on the project's dependency tree. To do so, the complete dependency graph of the libraries that are going to be updated (*updateGraph*), if developers add them as direct dependencies to the project, must be known to UPCY. Since the API of Maven Central does not give information about a library's (transitive) dependencies, we created a complete dependency graph for 8.35 million artifacts on Maven Central[1] using the graph database Neo4j. The database contains the (transitive) dependencies of each artifact, as well as their scopes. The database enables us to query libraries based on their properties (group, artifact, version) and their graph structure, in our case their dependencies, using the query language Cypher [26].

D. Identifying Compatible Updates

An incompatible update, if applied to the project, requires high maintenance by developers. Developers are forced to identify which (transitive) dependencies are affected by the incompatible update and have to adapt their own project's code. If the incompatibility cascades to an API call between dependencies, they also have to replace those since they usually do not have control over the code and pom.xml of these third-party dependencies. For instance, if the API calls that u invokes on t are binary incompatible with the updated version t', u also needs to be updated.

As a step towards reducing incompatible updates, UPCY computes a list of update steps and reports the API calls that suffer from source, binary, and (potential) semantic incompatibilities. To identify update steps, UPCY implements two approaches: First, for a naïve update, UPCY computes the number of incompatibilities that will occur if only the updated library is included as a direct dependency. Second, using the min-(s,t)-cut algorithm [24], UPCY explores complex update steps, which potentially involve the update of multiple libraries, that minimize the number of incompatibilities between the dependencies in the dependency graph.

E. Naïve Update

Similar to approaches such as Dependabot [12], UPCY simulates the transformation of the library update t to a direct dependency. In contrast to existing approaches, UPCY automatically computes changes in the dependency tree and resulting incompatibilities, and thus helps developers to spot regression by showing which API calls suffer from source, binary, or potentially from semantic incompatibility and must be adapted. To do so, UPCY queries the Neo4j database to get the *updateGraph* of the updated library t, as shown in Listing 1, and computes incompatibilities as described in detail in Section IV-H.

[1]At the time of writing: August 2022

F. Minimizing Incompatibilities using Min-(s,t)-Cuts

If the naïve update suffers from incompatibilities with other libraries, UPCY tries to identify alternative update steps for t using the unified dependency graph. In the project's unified dependency graph, all edges in this graph represent a dependency relation (cf. Section II). Further, we assume that each dependency relation also defines a use relation, as it is recommended practice to declare every used dependency also as a direct dependency [17]. Consequently, each edge also represents a compatibility requirement; the target library must be binary, source, and semantic, or blossom compatible depending on its position in the dependency graph. For the example graph in Figure 2, the edge from node $u3$ to t specifies that $u3$ depends on t, and $u3$ invokes t's API. Thus, the update t' must be binary compatible w.r.t. the API that $u3$ invokes. Note that the unified dependency graph may contain edges to superfluous, unused dependencies. In the future, we plan to prune such edges with the help of the call graph and also add usage relations not represented by dependency relations.

To identify update steps with a minimal number of incompatibilities between libraries, we need to separate the unified dependency graph into two partitions: one containing the project s, and one containing the library t, with a minimal number of edges crossing the two partitions. This can be optimally solved using the min-(s,t)-cut algorithm [24]. The min-(s,t)-cut algorithm computes for a graph and a pair of nodes (s, t) a cut of two separate partitions S and T with given nodes $s \in S$ and $t \in T$ that is minimal w.r.t. the weight of the edges crossing the partitions. Consequently, only nodes that are connected by the edges that are part of the cut must be ensured to be compatible with each other if they are updated. Since UPCY aims to minimize the number of incompatibilities between nodes, and each edge in the dependency graph represents one compatibility requirement, we assign all edges equal weight. The root nodes of the sink partition, which are the target of the edges on the cut, are thus candidates for updating by adding them as direct dependencies.

Crucially, simply computing the min-(s,t)-cut on the *directed* unified dependency graph does not minimize the incompatibilities to other dependencies as the resulting cuts ignore blossom, duplicate, and conflicting dependency compatibility. For a directed graph, like the dependency graph, a min-(s,t)-cut is equivalent to the maximum flow according to the max-flow min-cut theorem [24], thus edges to duplicate or conflicting dependencies located behind the updated library t, e.g., the duplicate w, are ignored as the directed edge from z to w is not part of a flow from s to t, and thus not part of the min-cut. For instance, computing the min-(s,t)-cut on the directed unified dependency graph in Figure 2 produces min-cuts of the weight 1, e.g., cutting the edge between s and $u2$. First, this min-cut ignores the blossom compatibility for u. Second, and more importantly, the cut ignores the compatibility edge between z and w.

To consider all compatibilities, UPCY computes the min-

(s,t)-cut on the undirected unified dependency graph with blossoms, e.g., $u1, u2, u3$ merged to blossom u. As a result, UPCY computes min-cuts of weight 2 for the graph in Figure 2; e.g., *min-cut 1* and *min-cut 2*.

For each found min-(s,t)-cut, UPCY tries to find compatible updates for the root nodes of the sink partition S by querying our Neo4j database of Maven Central for an *updateGraph*, as we explain in detail in the next section. If Neo4j returns a solution (a non-empty *updateGraph*), UPCY computes all incompatibilities w.r.t. the (updated) libraries in the *updateGraph*. For instance, for *min-cut 1*, UPCY checks if the calls between $s \rightarrow u2$ and $s \rightarrow x$ suffer from incompatibilities if $u2$ and x are updated, respectively, for *min-cut 2*, the calls $s \rightarrow u2$ and $x \rightarrow y$ if $u2$ and y are updated. UPCY stops when it finds a min-(s,t)-cut with zero incompatibilities or no further min-cuts can be found, and returns the list of dependency updates (*listUpdateSteps*) with the number of incompatibilities (cf. Algorithm 1). Finally, developers can add the dependencies in the computed update steps as direct dependencies to update the vulnerable or outdated library.

G. Querying the Graph Database of Maven Central using Cypher

For each found min-(s,t)-cut, UPCY queries our graph database of Maven Central for updates of the root nodes of the sink partition S to update the given library to the target version. In particular, UPCY asks Neo4j to (1) find new versions of the root nodes of the sink partition S that already depend on the target version of the library to update (2) that do not produce new incompatibilities – the (transitive) dependencies of the new versions of the root nodes should be compatible with each other, and (3) the (transitive) dependencies of those nodes (*updateGraph*). To avoid the introduction of new incompatibilities, UPCY asks in step (2) Neo4j to find versions of the root nodes that depend on the same version of a duplicate or conflicting library, e.g., the node w is included by transitive dependencies starting from $u2$ and x in the sink partition of *min-cut 1*. As an example, consider the min-cuts in Figure 2. For *min-cut 1*, UPCY tries to find (1) an update of the nodes $u2$ and x, where $u2$ depends on an updated version of t, (2) where $u2$ and x depend on the same version of the (transitive) dependency w, and (3) all (transitive) dependencies of the updates of $u2$ and x. For *min-cut 2*, UPCY tries to find (1) an update of $u2$ and y where $u2$ depends on an updated version of t, (2) $u2$ and y depend on the same version of the (transitive) dependency w, and (3) the (transitive) dependencies of the updates if $u2$ and y.

To find update steps for a computed min-(s,t)-cut, UPCY maps (1),(2), and (3) to Cypher, and then queries our graph database of Maven Central for libraries that fulfill the generated requirements. In the following, we present the Neo4j queries that UPCY creates on a conceptual, simplified level. Note that in UPCY's implementation, the queries are more complex and intertwined with each other to evaluate them together.

(1) Find versions of the sink's root nodes that already depend on the updated library: First, UPCY tries to find the target version of the library t in the Maven Central graph. To do so, UPCY generates for the library t in Figure 2 the Neo4j query shown in Listing 1. The query finds (`MATCH`) all libraries t with the group, artifact, version (GAV) *groupT:artifactT:targetVersionT*. Since the GAV is a unique identifier, Neo4j only returns a single library. To ease matching with the following queries (2) and (3), we relax the version to be greater or equal than the target version.

Listing 1
CYPHER QUERY: UPDATE OF LIBRARY

```
MATCH (t:MvnArtifact
    {group:"groupT",artifact:"artifactT",
    version>="targetVersionT"})
```

Second, UPCY tries to find new versions of the *root nodes* of the sink partition S that depend on the updated library t', if there is a path from the root node to t in the dependency graph. The root nodes are the libraries that are the targets of the edges cut by the computed min-(s,t)-cut. To do so, UPCY generates for every root node r a Neo4j query as shown in Listing 2. The query finds (`MATCH`) all libraries r, with the group and artifact, *groupR:artifactR*, that have a (transitive) dependency to the updated library t. The `*1..` property in the relation instructs Neo4j to search for paths of unlimited length (direct and transitive dependencies of r). If t itself is a root node r, UPCY skips the particular root node.

Listing 2
CYPHER QUERY: ROOT NODES OF SINK PARTITION

```
MATCH p=((r:MvnArtifact{group:"groupR",
    artifact:"artifactR"}) -[:DEPENDS_ON*1..]-> (t))
```

(2) Find compatible root nodes for duplicate and conflicting dependencies: If a sink partition has more than one root node, UPCY checks if duplicate or conflicting dependencies exist that are (transitively) included by two or more different root nodes. For instance, consider the dependency w in Figure 2. The dependency w is transitively included by the root nodes u and x in *min-cut 1*. No new incompatibilities should be introduced if u and x are updated. Thus, new versions of u and x should depend both on the same version of w.

To identify compatible updates of the root nodes, UPCY traverses the graph backward for all nodes in the sink partition to the *root nodes*. For *min-cut 1* and the root nodes u and x, UPCY yield the following results: $u \leftarrow u$, $u \leftarrow t$, $u \leftarrow t \leftarrow v$, $(u, x) \leftarrow (u, y) \leftarrow (t, z) \leftarrow w$, and so on. Analogously, for *min-cut 2*.

Then UPCY checks if two or more root nodes *share* a (transitive) dependency, e.g., x and u share the dependency w. We refer to those dependencies as a *shared dependency*, and UPCY creates additional queries. To ensure that an update is compatible, UPCY aims to find versions of u and x that

depend on the same version of w. To do so, UPCY creates for the shared node w the query shown in Listing 3. As an update of u or x may no longer be dependent on an instance of w, the matching is optional, defined by the relationship property `0...` If multiple solutions exist, only one solution is selected (`LIMIT 1`).

Listing 3
CYPHER QUERY: SHARED DEPENDENCIES

```
MATCH
 p1=((u:MvnArtifact) -[:DEPENDS_ON*0..]->
     (w:MvnArtifact)),
 p2 = ((x:MvnArtifact) -[:DEPENDS_ON*0..]-> (w))
 WHERE u.group="groupU" AND w.artifact="artifactU"
     AND w.group="groupW" AND w.artifact="artifactW"
 RETURN p1, p2 LIMIT 1
```

(3) Query for the update graph: Finally, UPCY asks Neo4j to return the complete *updateGraph* that are the root nodes as well as their (transitive) dependencies. To do so, UPCY generates for each root node r the Neo4j show in Listing 4. The found update graph is returned to UPCY.

Listing 4
CYPHER QUERY: UPDATE-GRAPH

```
MATCH p=((r)-[:DEPENDS_ON*1..]->(:MvnArtifact))
```

H. Compute Incompatibilities

As a result of the Cypher queries, UPCY receives the *updateGraph*, containing the library updates and their (transitive) dependencies, which are going to be incorporated into the project's dependency tree.

For each dependency in the *updateGraph*, UPCY checks if the dependency will have the shortest path in the updated dependency tree, and thus will shadow all other instances. If the dependency will become part of the project, UPCY checks all edges in the unified dependency graph that target the former instance of that dependency for incompatible API calls. To do so, UPCY queries SigTest [21] and SootDiff [27], for a list of API types and methods that are source or binary incompatible and whose method bodies changed - indicating potential semantic incompatibilities, and intersects them with the API calls in the unified dependency graph.

As an example, assume the naïve update t' of t will introduce a new version w'. UPCY checks if the API calls that u invokes on t are binary incompatible with t', and if the API calls from z to w' are compatible. To that end, UPCY creates the list of incompatible APIs for t, t' and w, w' using SigTest, and checks if the reported incompatible API types also occur as API calls in the unified dependency graph. If the dependency will not be part of the project because it is explicitly overridden by an older instance, UPCY reports a forward compatibility issue.

V. EVALUATION

A. Research Questions

To evaluate in how many cases UPCY can effectively support developers, we compare it to a naïve updating approach, which state-of-the-art tools apply.

In our first research question, we investigate in how many cases naïve updates fail, and more complex update steps as computed by UPCY are required: **RQ1: How often do naïve updates fail due to source code, binary or semantic incompatibilities?**

To assess the effect of dependency updates w.r.t. other libraries, UPCY simulates the changes an update causes in the dependency graph and tries to minimize the number of incompatibilities with other libraries. Thus, we set to measure how many dependencies a safe backward compatible update has to fulfill conflict, duplicate, blossom, binary, and source code compatibilities due to relations in the dependency graph. With an increasing number of compatibilities an update has to fulfill, the complexity for developers to reasons about an update increases along with the helpfulness of UPCY: **RQ2: How many compatibilities does an update has to fulfill (source code, binary, semantic, conflict, duplicate, and blossom)?**

To compare the effectiveness of UPCY to the naïve approach in practical environments, we measure how often our tool can provide optimized suggestions with fewer incompatibilities: **RQ3: In how many cases does UPCY minimize the number of incompatibilities compared to a naïve update?**

B. Study Objects & Methodology

Project Dataset: For the evaluation, we use the dataset of well-tested, open-source repositories that use Java as the primary language and Maven as the package manager sampled from GitHub created by Hejderup et al. [3]. The downloadable dataset [28] assembles commits of 462 different Maven projects. We were able to check out and build (`mvn compile`) 380 projects successfully. These 380 projects constitute in total 2,047 different Maven modules; 1,325 modules have a non-empty dependency graph. A Maven project typically assembles one more (independently) compilable subprojects, each with its own set of dependencies, so-called Maven modules.

The dataset is a representative sample of mid-sized, well-tested, open-source projects with a significant number of dependencies [3]. On average, each project assembles 668 methods, and 75% of the projects assemble around 588 or fewer declared methods. The median of direct dependencies is 7, and for transitive dependencies 16, indicating an expansion of transitive dependencies, which is in accordance with other recent studies [22], [16].

Creating Library Updates: For developers, it is crucial to update an outdated or vulnerable dependency quickly when a new vulnerability has been discovered to eliminate that dependency from the dependency tree.

By randomly choosing from each project up to 10 dependencies and then randomly up to 10 newer versions to seed such

updates, we derive an adequate test set for comparing the naïve and UPCY's approach, e.g., in cases where a vulnerability has been disclosed.

In total, we created 29,698 such updates for 5,558 different libraries in 1,325 modules: 8,327 updates of direct dependencies, and 21,371 (71.96%) updates of transitive dependencies. On average, we generated 22 (mean, 11.3 std) updates per module, with a positively skewed distribution. 75% of all modules in our sample had around 32 or fewer updates. The largest project had 85 updates.

We only selected dependencies with the scope *compile* and excluded dependencies with the scope *testing, system, runtime, provided* since they are usually unavailable during compile-time and often do not specify a version, and thus cannot be statically checked.

C. Results

RQ1: Effectiveness of naïve updates: State-of-the-art tools apply a naïve approach to update libraries by transforming the library to a direct dependency and set it to the new version in the pom.xml. We performed each update as a naïve update to evaluate the effectiveness of this approach. To check if a naïve update introduced source code, and binary or semantic incompatibilities, we executed the Maven commands `mvn compile` and `mvn test` two times: on the original project and after applying the naïve update.

If the `mvn compile` command fails during the second invocation, the new library version introduced source code incompatibilities, as the project's code that invokes the updated library's API can no longer be compiled successfully. Because it is recommended practice to include libraries whose API is invoked as direct dependencies, compile failures usually occur for direct dependencies only. If the `mvn test` command fails during the second invocation, the new library version introduced binary or semantic incompatibilities as formerly valid test cases fail to link or run successfully with the updated library.

Table I shows the results. 26,966 (90%) of the 29,698 updates compiled successfully and no tests failed. Only 2,732 updates actually result in compile (1,393) or test failures (1,339), shown in column *#total*. The compile and test failures occurred in 372 (28.07%) of the 1,325 modules. This is not surprising: Since the 29,698 updates only affect 5,558 different libraries, the majority of updates are version increments of the same libraries in the same modules. Consequently, these results indicate that the likelihood is high that if the update of a library from version 1.1 to version 1.1.5 runs successfully, updating that library to version 1.2 and version 1.4 will also run successfully.

Note that Hejderup et al. [3] found that test cases of a module typically cover less than 60% of their function calls to direct dependencies, and the coverage drops to 21% for calls to transitive dependencies, which was the majority of updates in our dataset (71.96%). Since we execute `mvn test` to detect binary incompatibilities and semantic breaking changes, the number of failed updates is a **lower bound** only.

TABLE I
STATISTICS FOR FAILED UPDATES

failure type	#total	per module				
		mean	std	min	median	max
build or test	2,732	7.34	8.11	1	6	36
build	1,393	5.71	7.34	1	4	36
test	1,339	7.52	8.03	1	6	36

Findings from RQ1: 28.07% of the 1,325 modules suffered from compile or test failures when applying naïve updates, which are implemented by state-of-the-art tools.

RQ2: Complexity of Library Updates - binary, semantic, conflict, and blossom incompatibilities: To conduct a safe backward compatible update, developers have to ensure that no incompatibilities occur between the updated library, the project, and other libraries in the dependency tree. To estimate how complex this reasoning is, we computed how many binary and potential semantic incompatibilities an update introduces, and with how many other libraries compatibility must be ensured.

To compute binary (ABI) incompatibilities between two versions of a library, we used Oracle's SigTest tool [21], [29], which is specifically designed to compare the signatures of two versions of the same library and report binary incompatibilities.

To get an estimate of potential semantic incompatibilities, we compare the bytecode of the former method's body with the new bytecode using the tool SootDiff [27]. SootDiff's comparison was specifically designed to be resistant to changes induced by various compilation schemes, and thus allows us to check for bytecode equivalence even if different compilers, Java versions, or source code changes have been applied to one of the classes. If SootDiff finds a difference, this indicates that the method's semantics *may* have changed. Nevertheless, only checking the bodies of API methods will cause transitive changes in the preconditions to be missed. For instance, if a public method m_{api} calls a private method m_{priv}, and only m_{priv} has changed, we would miss the fact that m_{api}'s API has changed if m_{priv} is excluded from the comparison [30]. Thus, we build the library's call graph using Soot's CHA implementation and iterate over the call-chain of each API method and check if the body of any method along the chain has changed [30]. This approach yields an over-approximation: SootDiff will detect and report a difference if a (potential) semantic change exists. Note that the precise detection of semantic changes is impossible, as described in Section III.

Figure 3 presents the results for binary and potential semantic incompatibilities in the form of a violin plot, using a log transformation with base 10 to deal with outliers. Overall, 65.38% of all library updates suffer from binary incompatibilities issues. On average, an update introduces 83

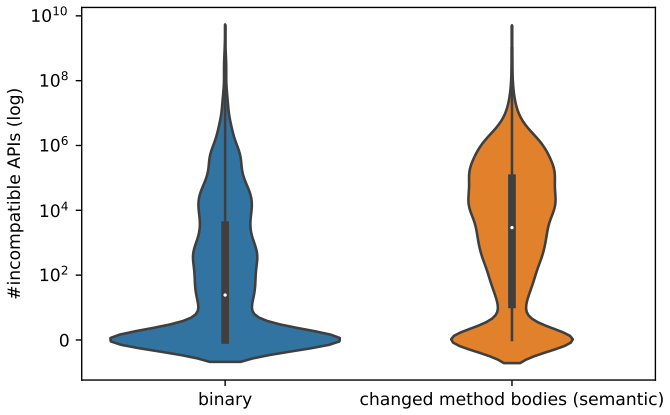

Fig. 3. Incompatibilities in Updates

TABLE II
BLOSSOM, CONFLICT, DUPLICATE, AND BINARY COMPATIBILITY FOR AN UPDATE.

dependency	mean	std	min	median	max
size of blossoms	4.19	3.92	1	2	30
#conflicts	1.71	1.73	1	1	16
#duplicates	2.53	3.12	1	1	31
#binary dependents	1.95	2.33	1	2	32

binary incompatibilities, and 75% of all libraries contain fewer or equal than 35 binary incompatibilities.

The distribution of potentially semantic incompatibilities (changed method bodies) suggests two classes of libraries. In the first class - the peak at 0, we have library updates that do not introduce any changes in existing method bodies. In the second class - the peak at 127, we have library updates that contain changes in more than 100 methods, indicating larger refactorings.

To compute to how many other libraries a library has to be binary and semantic compatible, we computed the library's connectivity in the dependency graph. As described in Section IV-F, all edges in the unified dependency graph represent a dependency and use relation between two dependencies, thus the connectivity of a library shows to how many other libraries compatibility must be ensured when updating. Table II shows the results. A developer has, on average, to ensure binary compatibility to 1.9 other libraries in the dependency graph, and consider 1.7 conflicts and 2.5 duplicates. If a library is part of a blossom, the blossom contains 4 libraries on average. Thus, developers must not only reason about the relations between their project and the library but also respect duplicate and conflicting dependencies as they can induce inconsistent runtime behavior [1].

Findings from RQ2: When updating a library, on average, developers need to maintain binary compatibility to at least 2 further libraries, and consider conflicts with 1.7 other instances of that library.

TABLE III
COMPARISON UPCY VS NAÏVE UPDATES.

update	count	mean	std	min	median	max
naïve	3,821	2.17	5.38	1	1	124
UPCY	3,821	1.15	1.29	0	1	15
fewer incompatibilities	1,572	0.62	1.38	0	0	15
more incompatibilities	14	2.36	0.49	2	2	3
incompatibility reduction		1.01	5.29	-2	0	124
complexity of min-(s,t)-cut		1.63	0.98	0	2	10

RQ3: Minimization of Incompatibilities by UPCY*:* To evaluate the effectiveness of UPCY, we executed it on the 20,610 updates of transitive dependencies. To increase the performance of the Neo4j database lookups, we restricted the length for transitive dependencies to 5 and set a timeout of 180sec for a query. Thus, UPCY may miss update options in complex scenarios in which shared nodes have a path length greater than 5 or run exceptionally complex queries.

In total, UPCY could successfully compute for 16,884 (81.9%) of 20,610 updates if incompatibilities exist using SigTest and SootDiff. For the other updates, the computation of incompatibilities was unsuccessful since either the compilation process of the unmodified module failed, the call graph was empty, e.g., for test projects, or the tools SigTest failed.

Table III shows the descriptive statistics for the update suggestions that on the UPCY computed using its min-(s,t)-cut approach. Note that the table shows the number of libraries to which the update introduces incompatibilities. It does not give the absolute number of violated API calls.

The table shows that 3,821 of the naïve updates produced incompatibilities, and thus UPCY computed alternative updates. The mean in row UPCY shows that the updates that UPCY produces on average have fewer incompatibilities than naïve updates. In only 14 cases, the min-(s,t)-cut update suggestions had more incompatibilities than the naïve update, shown in row *more incompatibilities*, resulting in the negative min value in row *reduction*.

For 1,572 (41.1%) updates, UPCY computed a min-(s,t)-cut update with fewer incompatibilities. 1,102 (70.1%) of these updates have zero incompatibilities. In cases in which the naïve update produced incompatibilities, UPCY computed update steps that require more than one library to update (min-(s,t)-cut complexity in Table III) or found a compatible update of a preceding dependency in the dependency graph. The high standard deviation of the naïve updates shows that the number of libraries to which an update introduces incompatibilities heavily varies. The standard deviation for UPCY is lower; UPCY consistently reduces incompatibilities. While the number of updates for which UPCY computed a min-(s,t)-cut seems low at first glance, this is not surprising given the insights from **RQ1**: the majority of updates are

version increments of the same library, and the numbers show that the likelihood is high that if a library update succeeded, further increments of the version, i.e., another naïve update, will succeed, too.

On average, the updates computed by UPCY's min-(s,t)-cut approach require to update 1.63 (mean - in row *complexity of min-(s,t)-cut*) different libraries. The biggest min-(s,t)-cut that minimizes the incompatibilities requires the update of 10 libraries.

Figure 4 shows a simple example of the computed min-(s,t)-cut for the dependency graph of the project *mybatis-shards* in the dataset, requiring the update of two dependencies to maintain compatibility. The outdated library in the example is cglib:cglib:2.2.2. The library should be updated to version 3.3.0. Instead of simply updating cglib to the target version, which is the naïve approach, UPCY computed the min-cut cutting the edges between the project and the target library cglib:cglib and asm:asm:3.3.1. Consequently, the min-cut shows that both libraries cglib and asm needs to be updated. Using the graph database of Maven Central, UPCY found that the target version cglib:cglib:3.3.0 depends on asm:7.1 and returns both in the *updateGraph*.

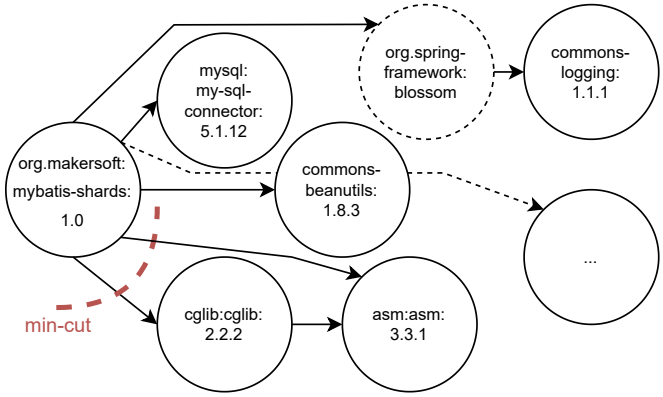

Fig. 4. Example: Min-(s,t)-Cut in project mybatis-shards updating cglib

The results show that UPCY can effectively find update suggestions with fewer incompatibilities than naïve updates. While other tools, which only apply naïve updates, yield incompatibilities, UPCY can successfully recommend compatible updates even if multiple libraries need to be updated to achieve. As the dataset from Hejderup [3] exclusively contains well-tested, open-source GitHub projects, the results show that the need for updating multiple libraries to achieve compatibility is prevalent in practical cases.

Findings from RQ3: UPCY can effectively reduce incompatibilities for 41.1% of the updates in which the naïve update lead to source or binary incompatibilities.

Required Computation Resources for UPCY*:* The computation resources for UPCY are distinguished into resources required for creating the Neo4j database of Maven Central, which is only done once, and resources required for computing a list of updates, which is done for a dependency update. We created the Maven graph in Neo4j on a machine with 68GB RAM and 12vCPUs within 10-11 days. The database has a size of 22 GB.

For generating the list of updates UPCY executes two steps: computing the min-cut, and querying Neo4j. The min-(s,t)-cut computation takes ms only on the developer's machine. The time for querying the Maven dependency graph depends on the machine hosting the database. In our case, queries ranged from ms to a few minutes, depending on the number of dependencies to update and the size of the min-cut. However, the query performance can be further optimized, e.g., by using indices or by splitting large queries into smaller ones. Neo4j can handle millions of nodes; the Maven Central graph is relatively small with 8.5 million nodes.

In our evaluation, we set a timeout limit of 180 sec per Cypher query. Due to the timeout, the project *cassiomolin/jersey-jwt* failed to complete.

VI. THREATS TO VALIDITY

Sampling libraries and versions randomly for creating updates poses a threat to our results, as we may select libraries that are either updated relatively rarely or frequently. To mitigate this risk, we used the sampled dataset from Hejderup [3] of mid-sized, well-tested, open-source projects from GitHub covering 5,558 different libraries in 1,325 Maven modules.

A threat to *RQ1* and *RQ3* is the fact that some of the generated updates ask to update the selected library to the latest release. We found that for some dependent libraries (that are the ones preceding the library to update), no versions have been published yet that depend on the latest release.

Similarly, some generated updates ask to update to a release that has been skipped by dependent libraries. For instance, we generated an update step from org.slf4j:slf4j-api:1.7.21 to version 1.7.34 with predecessor logback-classic:1.17. There exists no version of logback-classic that depends on slf4j-api:1.7.34. Instead, recent versions depend on newer releases of slf4j-api. In these cases, UPCY could only identify naïve updates.

The biggest threat for *RQ3* is the completeness of our database. In cases where our database is incomplete, UPCY cannot identify valid update steps, making *RQ3* an under-approximation.

A further threat is that UPCY does not compute all min-(s,t)-cuts for the unified dependency graph. Although Nagamochi et al. [31] give an algorithm that finds all minimum cuts efficiently, there are no published implementations of the algorithm [32], and we refrain from implementing the algorithm ourselves. Nevertheless, our evaluation shows that UPCY can identify improved updates even if not all min-(s,t)-cuts are considered.

VII. Related Work

A. Studies: How Developers Update Dependencies

Researchers have conducted several studies investigating practices around updating (vulnerable) open-source dependencies [6], [9], [10], [4], [2]. The studies show that developers typically hesitate to update libraries since they are afraid of introducing breaking changes and refactoring efforts, and thus prioritize other tasks. Consequently, Kula et al. [6] found for 81.5% of the studied dependencies that developers did not update those dependencies even though they contain known vulnerabilities.

Mirhosseini et al. [9] found that automated pull requests can encourage developers to update dependencies quicker, but they also suffer from high rates of rollbacks and gaps in continuous integration, deferring developers from automatically updating. Similarly, Bogart et al. [10] point out that developers perceive automated pull requests as information overload, which do not help to evaluate the impact of a specific update.

B. Update Compatibility Analysis

Dietrich et al. [15] studied the effect of source code and binary compatibility issues in the Qualitas corpus. They found that 75% of library updates introduced breaking changes, but only a few resulted in errors. The authors emphasize that their study only applies to the Qualitas corpus, which only has a few projects with intertwined libraries. Wang et al. [1] conducted a study on whether dependency conflicts may break a project's semantics. The study finds that dependency updates introduce semantic breaking changes in 50 of the 128 sample Java projects. To detect semantic breaking changes between two versions, the authors present SENSOR. SENSOR applies call graph analysis to detect changed API and applies automated test generation to uncover semantic breaking changes.

Hejderup et al. [3] empirically investigate if test suits, which are used by automatic pull requests, are reliable in detecting breaking changes in updates. The study finds that test suits can only cover 58% of direct and 21% of transitive dependency calls, and thus are not reliable. To alleviate this situation, the authors developed a static change impact analysis Uppdatera. Uppdatera statically computes the difference between two versions of a library based on the source codes' abstract syntax tree to identify methods with code changes. Using a heuristic Uppdatera determines if the changes break the method's semantics. Then, Uppdatera computes the call graph for the project and dependencies and evaluates if the project invokes methods with breaking changes.

VIII. Conclusion

In this paper, we present UPCY - an approach to provide suggestions for dependency updates. As recent research highlights, developers hesitate to update dependencies and mistrust automated approaches, like Dependabot, since they are afraid of introducing incompatibilities that break their projects or lead to unwanted side effects. Thus, our approach aims to support developers in finding updates with minimal incompatibilities to other dependencies. To do so, UPCY investigates the project's dependency graph and explores multiple update options, whereas state-of-the-art approaches naïvely focus on the outdated library exclusively. UPCY uses the min-(s,t)-cut algorithm to identify updates with minimal incompatibilities, and queries a graph database of Maven Central for new dependencies. To assess the impact of an update, UPCY determines incompatible API calls using the static analysis framework Soot. As an output, UPCY proposes a list of dependencies that developers can add as direct dependencies to eliminate a vulnerable or outdated dependency (especially a transitive one) from their project's dependency graph.

We evaluated the naïve approach against UPCY on a representative dataset of 1,325 well-tested, open-source Java projects from GitHub. Our results show that UPCY can effectively provide update suggestions that produce fewer incompatibilities than current, naïve approaches. In 41.1% of the cases in which the naïve update leads to incompatibilities, UPCY could detect an update option with fewer incompatibilities to other libraries, where 70.1% of the generated updates even have zero incompatibilities.

We implemented UPCY for Java and its package manager Maven as examples in this work. However, UPCY's approach can be adapted to other programming languages and package managers as well.

UPCY can be directly applied to package managers with a global dependency graph and use similar conflict resolution mechanisms as Maven, e.g., Python's package manager pip also permits only a single, non-conflicting version of a dependency in a project. For package managers like node.js that maintain a complete dependency graph per dependency, permitting multiple conflicting versions, UPCY's algorithm needs to be adapted respecting that resolution mechanism.

In future work, a user study checking the understandability of the reported violations and the acceptance of UPCY's update steps is beneficial. Further, we aim to improve the assessment of incompatibilities by incorporating more advanced techniques for impact assessment, for instance, the approach provided by Hejderup et al. [3]. Our approach can be easily adapted to minimize other metrics as well, e.g., decreasing the number of violated API calls by using the number of calls as an edge weight in the graph.

Data Availability

We have made the source code of UPCY, the evaluation pipeline, and our data publicly available on GitHub https://github.com/secure-software-engineering/upcy and Zenodo https://doi.org/10.5281/zenodo.7037673.

Acknowledgments

This research was partially funded through the project Automated risk analysis with respect to open source-dependencies (Hektor) by the German Research Foundation (DFG), under grant number 160364472.

References

[1] Y. Wang, R. Wu, C. Wang, M. Wen, Y. Liu, S. Cheung, H. Yu, C. Xu, and Z. Zhu, "Will dependency conflicts affect my program's semantics?" *IEEE Transactions on Software Engineering*, vol. 48, no. 07, pp. 2295–2316, jul 2022. [Online]. Available: https://doi.org/10.1109/TSE.2021.3057767

[2] J. Düsing and B. Hermann, "Analyzing the direct and transitive impact of vulnerabilities onto different artifact repositories," *Digital Threats*, vol. 3, no. 4, feb 2022. [Online]. Available: https://doi.org/10.1145/3472811

[3] J. Hejderup and G. Gousios, "Can we trust tests to automate dependency updates? a case study of java projects," *Journal of Systems and Software*, vol. 183, p. 111097, 2022. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0164121221001941

[4] G. A. A. Prana, A. Sharma, L. K. Shar, D. Foo, A. E. Santosa, A. Sharma, and D. Lo, "Out of sight, out of mind? How vulnerable dependencies affect open-source projects," *Empirical Software Engineering*, vol. 26, no. 4, 2021. [Online]. Available: https://doi.org/10.1007/s10664-021-09959-3

[5] I. Pashchenko, D.-L. Vu, and F. Massacci, "A qualitative study of dependency management and its security implications," in *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 1513–1531. [Online]. Available: https://doi.org/10.1145/3372297.3417232

[6] R. G. Kula, D. M. German, A. Ouni, T. Ishio, and K. Inoue, "Do developers update their library dependencies?" *Empirical Software Engineering*, vol. 23, no. 1, pp. 384–417, 2018. [Online]. Available: https://doi.org/10.1007/s10664-017-9521-5

[7] Y. Wang, B. Chen, K. Huang, B. Shi, C. Xu, X. Peng, Y. Wu, and Y. Liu, "An Empirical Study of Usages, Updates and Risks of Third-Party Libraries in Java Projects," *Proceedings - 2020 IEEE International Conference on Software Maintenance and Evolution, ICSME 2020*, pp. 35–45, 2020. [Online]. Available: https://doi.org/10.1109/ICSME46990.2020.00014

[8] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An empirical study of third-party library updatability on android," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. New York, NY, USA: Association for Computing Machinery, 2017, p. 2187–2200. [Online]. Available: https://doi.org/10.1145/3133956.3134059

[9] S. Mirhosseini and C. Parnin, "Can automated pull requests encourage software developers to upgrade out-of-date dependencies?" *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pp. 84–94, 2017. [Online]. Available: https://dl.acm.org/citation.cfm?id=3155577

[10] C. Bogart, C. Kästner, and J. Herbsleb, "When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, 2015, pp. 86–89. [Online]. Available: https://doi.org/10.1109/ASEW.2015.21

[11] Greenkeeper. 2022-08-26. [Online]. Available: https://greenkeeper.io/

[12] Dependabot. 2022-08-26. [Online]. Available: https://dependabot.com/

[13] Renovate. 2022-08-26. [Online]. Available: https://www.whitesourcesoftware.com/free-developer-tools/renovate

[14] C. Bogart, C. Kästner, J. Herbsleb, and F. Thung, "How to break an api: Cost negotiation and community values in three software ecosystems," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, p. 109–120. [Online]. Available: https://doi.org/10.1145/2950290.2950325

[15] J. Dietrich, K. Jezek, and P. Brada, "Broken promises: An empirical study into evolution problems in Java programs caused by library upgrades," *2014 Software Evolution Week - IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering, CSMR-WCRE 2014 - Proceedings*, pp. 64–73, 2014. [Online]. Available: https://doi.org/10.1109/CSMR-WCRE.2014.6747226

[16] 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*, pp. 1–1, 2021. [Online]. Available: https://doi.org/110.1109/TSE.2021.3101739

[17] Introduction to the dependency mechanism. 2022-08-26. [Online]. Available: https://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html

[18] K. Huang, B. Chen, B. Shi, Y. Wang, C. Xu, and X. Peng, "Interactive, effort-aware library version harmonization," *ESEC/FSE 2020 - Proceedings of the 28th ACM Joint Meeting European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 518–529, 2020. [Online]. Available: http://doi.acm.org/10.1145/3368089.3409689

[19] Evolving java-based apis. 2022-08-26. [Online]. Available: https://wiki.eclipse.org/Evolving_Java-based_APIs

[20] Javase specification. 2022-08-26. [Online]. Available: https://docs.oracle.com/javase/specs/jls/se8/html/

[21] Sigtest github repository. 2022-08-26. [Online]. Available: https://github.com/jtulach/netbeans-apitest

[22] 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*, ser. ESEM '18. New York, NY, USA: ACM, 2018, pp. 42:1—-42:10. [Online]. Available: http://doi.acm.org/10.1145/3239235.3268920

[23] 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*, ser. CASCON '10. USA: IBM Corp., 2010, p. 214–224. [Online]. Available: https://doi.org/10.1145/1925805.1925818

[24] L. R. Ford and D. R. Fulkerson, "Maximal Flow Through a Network," *Canadian Journal of Mathematics*, vol. 8, pp. 399–404, nov 1956. [Online]. Available: https://www.cambridge.org/core/product/identifier/S0008414X00036890/type/journal_article

[25] depgraph-maven-plugin. 2022-08-26. [Online]. Available: https://github.com/ferstl/depgraph-maven-plugin

[26] Cypher query language. 2022-08-26. [Online]. Available: https://neo4j.com/developer/cypher/

[27] A. Dann, B. Hermann, and E. Bodden, "Sootdiff: Bytecode comparison across different java compilers," in *Proceedings of the 8th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, ser. SOAP 2019. New York, NY, USA: Association for Computing Machinery, 2019, p. 14–19. [Online]. Available: https://doi.org/10.1145/3315568.3329966

[28] Dataset. can we trust tests to automate dependency updates? a case study of java projects. 2022-08-26. [Online]. Available: https://zenodo.org/record/4479015/#.Yw0Bg-xBzUa

[29] Oracle - sigtest user guide. 2022-08-26. [Online]. Available: tps://docs.oracle.com/javacomponents/sigtest-3-1/user-guide/i

[30] D. Foo, H. Chua, J. Yeo, M. Y. Ang, and A. Sharma, "Efficient static checking of library updates," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ser. ESEC/FSE 2018. New York, NY, USA: Association for Computing Machinery, 2018, p. 791–796. [Online]. Available: https://doi.org/10.1145/3236024.3275535

[31] H. Nagamochi, Y. Nakao, and T. Ibaraki, "A fast algorithm for cactus representations of minimum cuts," *Japan Journal of Industrial and Applied Mathematics*, vol. 17, no. 2, p. 245, 2000. [Online]. Available: https://doi.org/10.1007/BF03167346

[32] M. Henzinger, A. Noe, C. Schulz, and D. Strash, "Finding All Global Minimum Cuts in Practice," in *28th Annual European Symposium on Algorithms (ESA 2020)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), F. Grandoni, G. Herman, and P. Sanders, Eds., vol. 173. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2020, pp. 59:1–59:20. [Online]. Available: https://drops.dagstuhl.de/opus/volltexte/2020/12925