Can we trust tests to automate dependency updates? A case study of Java Projects

Hejderup, Joseph; Gousios, Georgios

DOI
10.1016/j.jss.2021.111097

Publication date
2022

Document Version
Final published version

Published in
Journal of Systems and Software

Citation (APA)
Hejderup, J., & Gousios, G. (2022). Can we trust tests to automate dependency updates? A case study of Java Projects. *Journal of Systems and Software*, *183*, 1-13. Article 111097. https://doi.org/10.1016/j.jss.2021.111097

The Journal of Systems & Software 183 (2022) 111097

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Can we trust tests to automate dependency updates? A case study of Java Projects☆

Joseph Hejderup *, Georgios Gousios

Delft University of Technology, Van Mourik Broekmanweg 6, 2628 XE, Delft, The Netherlands

ARTICLE INFO

Article history:
Received 16 February 2021
Received in revised form 30 July 2021
Accepted 10 September 2021
Available online 24 September 2021

Keywords:
Semantic versioning
Library updates
Package management
Dependency management
Software migration

ABSTRACT

Developers are increasingly using services such as Dependabot to automate dependency updates. However, recent research has shown that developers perceive such services as unreliable, as they heavily rely on test coverage to detect conflicts in updates. To understand the prevalence of tests exercising dependencies, we calculate the test coverage of direct and indirect uses of dependencies in 521 well-tested Java projects. We find that tests only cover 58% of direct and 21% of transitive dependency calls. By creating 1,122,420 artificial updates with simple faults covering all dependency usages in 262 projects, we measure the effectiveness of test suites in detecting semantic faults in dependencies; we find that tests can only detect 47% of direct and 35% of indirect artificial faults on average. To increase reliability, we investigate the use of change impact analysis as a means of reducing false negatives; on average, our tool can uncover 74% of injected faults in direct dependencies and 64% for transitive dependencies, nearly two times more than test suites. We then apply our tool in 22 real-world dependency updates, where it identifies three semantically conflicting cases and three cases of unused dependencies that tests were unable to detect. Our findings indicate that the combination of static and dynamic analysis should be a requirement for future dependency updating systems.

1. Introduction

Modern package managers facilitate reuse of open source software libraries by enabling applications to declare them as versioned dependencies. Crucially, when a new version of a dependency is made available, package managers will automatically make it available to the client application. This mechanism helps projects stay up-to-date with upstream developments, such as performance improvements or bug fixes, with minimal fuss. Typically, package managers implement a set of interval operators (dependency version ranges) on top of the SemVer protocol (npm, 2018) that developers use to declare update constraints. For example, a dependency declared with the range >= 1.0.0 < 1.5.0 restricts updates to backward-compatible changes up to 1.5.0. On the other hand, >= 1.0.0 welcomes automatic updates of all new version releases starting from 1.0.0. Given a new library release with version 1.5.0, the latter constraint will allow an update but the former will not.

In practice, most package managers use a liberally interpreted version of the SemVer protocol with no vetting, allowing library maintainers to release new changes based on their self-interpretation of backward compatibility (npm, 2018; Bogart et al., 2016). As a consequence, client programs may unexpectedly discover regression-inducing changes, such as bugs or semantic changes that break code contracts. Discovering, debugging and resolving such issues, as exemplified in Fig. 1, remains a challenging task for development teams (Bogart et al., 2016). In fact, unexpected regressions are one of the main reasons that deter developers from upgrading dependencies to new versions (Kula et al., 2018).

Developers can mitigate the risk of integration errors by either using restrictive strategies, such as *version locking*, or permissive strategies involving *dependency update tooling*. Version locking effectively makes the dependency tree of client programs immutable and disables automated updates. This strategy offers maximum stability but is prone to incurring technical debt due to outdated dependencies. Moreover, developers need to manually discover and apply security hotfixes. On the other hand, dependency update checkers analyze version compatibility before deciding to update. There are two main techniques for deciding version compatibility, *breaking change detection* (Mezzetti et al., 2018; Brito et al., 2018; Foo et al., 2018b) and *regression testing* (Agrawal et al., 1995; Cleve and Zeller, 2005). Detecting potential breaking changes (i.e., backward API incompatibilities) prevents client programs from updating to versions that will result in compile failures. A major shortcoming of this technique

☆ Editor: Aldeida Aleti.

* Corresponding author.
E-mail addresses: j.i.hejderup@tudelft.nl (J. Hejderup), g.gousios@tudelft.nl (G. Gousios).

https://doi.org/10.1016/j.jss.2021.111097

is that it depends on the compilation and the existence of a static type system; many of today's most popular languages are dynamically typed. A more popular option among developers is the use of services providing automated dependency updating, such as `greenkeeper.io` (greenkeeper.io, 2019), DEPENDABOT (Dependabot, 2019), and `renovate` (Renovate, 2019), that use project test suites to detect regression changes on every new update.

The effectiveness of such services depends highly on the quality of end-users test suites (Inozemtseva and Holmes, 2014). Poor test coverage of dependency usage in client code can lead to missing update-induced regressions. Recent studies (Hilton et al., 2018; Kochhar et al., 2017) suggest that high statement coverage in test suites does not guarantee to find regressions in code changes. Failing to detect regressions stemming from updates can have dire consequences for client programs: for example, users dependent on NPM's `event-stream` package did not notice a malicious maintainer planting a hidden backdoor for stealing bitcoin wallets inside the library's source code (npm Inc., 2018). Moreover, a recent qualitative study (Mirhosseini and Parnin, 2017) also revealed that developers are generally suspicious of automatically updating their dependencies. One of the prime reasons is that developers perceive their tests as unreliable. To reduce the number of false negative updates, we develop a static change impact analysis for dependency updates called UPPDATERA. By statically identifying changed functions and approximating call-relationships between an application and its dependencies, change impact analysis can fill in gaps where test suites have limited coverage or cannot reach.

In this paper, we set out to empirically understand how reliable developer tests are in automated dependency updating by addressing the following research questions:

- **RQ1:** *Do test suites cover the uses of third-party libraries in projects?*
- **RQ2:** *How effective are project test suites and change impact analysis in detecting semantic changes in third-party library updates?*
- **RQ3:** *How useful is static analysis in complementing tests for compatibility checking of new library versions?*

To study the prevalence of tests exercising dependencies in projects, we first establish all uses of library functionality from direct and transitive dependencies in 521 well-tested projects and then measure how much test suites cover those usages. By systematically mutating dependency uses in 262 projects, we then conduct a comparative study on the adequacy of test suites and change impact analysis in detecting artificial updates with simple faults. To understand the strengths and weaknesses of using static analysis as a complement to tests in a practical setting, we evaluate the performance of test suites and UPPDATERA on 22 newly created pull requests that update dependencies.

Our results indicate that tests lack considerable coverage of function calls in projects that target library dependencies; average coverage is 58% for direct dependencies and 21% for transitive dependencies. Similarly, the average effectiveness of test suites is 47% for direct dependencies and 35% for transitive dependencies. When using change impact analysis, the average effectiveness increases to 74% for direct dependencies and 64% for transitive dependencies, suggesting that static analysis can cover open coverage gaps in tests. Through our manual analysis, UPPDATERA was able to catch three unsafe updates and three unused dependencies, suggesting that it is potentially more effective in avoiding faulty updates than tests. However, it is also more prone to false positives due to difficulties in evaluating over-approximated execution paths.

Our findings raise awareness of the risks involved with automated dependency updating. Tool creators should consider reporting how adequately project tests exercise changed functionality in libraries under update. In future updating systems, tool creators should investigate hybrid workflows to complement gaps in regression testing with static analysis and help developers with prioritizing testing efforts.

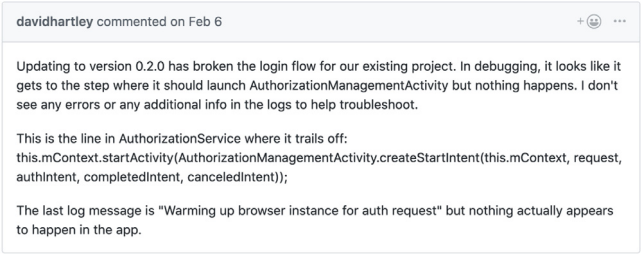

Fig. 1. Update failure in `okta/okta-sdk-appauth-android`, #81.

2. Background

2.1. Package managers

Package managers such as Java's MAVEN, JavaScript's NPM, or Rust's CARGO provide tooling to simplify the complexities of maintaining, distributing, and importing external third-party software libraries in development projects. As a community service to its users, package managers also host a public Online Package Repository (OPR) where developers can freely contribute with new packages (e.g., a database driver) or build upon existing packages (e.g., use a parser library to build a JSON parser). This helps package manager users to reduce development efforts by benefitting from existing functionality in their language environments. In a nutshell, a package is a distributable, versioned software library.

Because of the relative ease of building packages on top of each other, OPRs today grow quickly and become evermore interdependent (Decan et al., 2017; Kikas et al., 2017). As a consequence, package manager users experience a dynamic growth of new hidden dependency imports in their projects and frequent dependency updates that increase the risk of build failures due to breaking backward compatibility (Decan et al., 2018b,a; Wittern et al., 2016; Bogart et al., 2016). The risk of breaking backward compatibility varies between OPRs: NPM and MAVEN CENTRAL move the burden of checking incompatible changes on its users while R/CRAN minimize this risk by requiring a mutual change-cost negotiation between library maintainers and their users (Bogart et al., 2016). Users of OPRs, such as NPM or MAVEN, either uses additional tooling or disable dependency updates through version-locking as a protective measure. Version-locking dependencies guarantee a stable build environment. Additional tooling provides an extra layer control by scanning dependencies for vulnerabilities (Decan et al., 2018a), freshness (Cox et al., 2015) or update compatibility (Mezzetti et al., 2018; Foo et al., 2018b).

2.2. Safe backward compatible updates

Update checkers such as *cargo-crusador* (Anderson, 2018), *JAPICC* (Ponomarenko, 2011), and *dont-break* (Bahmutov, 2014) typically determine backward compatibility by ensuring that the new version is consistent with the public API contract of the old version. Removals or changes in method signatures, access modifiers, and types (e.g., classes and interfaces) are examples of inconsistencies that can lead to compile failures in client code (Dietrich et al., 2014; Raemaekers et al., 2017).

```
1  package client {
2      import p2.B;
3      class Main {
4          void int main {B.b(); B.z();}
5      }
6  }
7  package p2 {
8      import p1.A;
9      class B {
10         int b() {
11             int y = 1;
12             if A.v(y){y+2;}
13             int x = A.a();
14             if x > 0 {return 0;}
15             return x + y;
16         }
17         //...
18   -     bool z() {return false;}
19   +     bool z() {return make_false();}
20         //...
21   +     bool make_false() {return false;}
22     }
23 }
24 package p1 {
25     class A {
26         //...
27   -     int a() {return 0;}
28   +     int a() {return 1;}
29         //...
30   -     bool v(int a) {a > 0 ? true : false}
31   +     bool v(int a) {a == 0 ? true : false}
32     }
33 }
```

Listing 1: Changes in dependencies that break client semantics

Checking dependency updates for API inconsistencies is a necessary precondition to a safe update, but not a sufficient one. From Listing 1, we consider an additional class of changes, *semantic changes*, that are API-compatible (i.e., respects the public API contract) but introduces incompatible behavior (i.e., regression changes) for clients after dependency updates. The code example illustrates a `client` that depends on `p2` which in turn depends on `p1`. There are two changes that are not semantic preserving in `p1`: `a()` returns 1 instead of 0 (line 27–28) and `v(int a)` compares variable `a` with a different comparison operator (line 30–31). On the other hand, the change in `p2` is semantic preserving: `z()` still returns `false` despite replacing it with a method call to `make_false` (line 18–21). Given a scenario in which `client` automatically updates to the next release of `p1`, and `p1` updates to the next release of `p2`. The changes made in `p1` will indirectly impact the behavior of `client` despite seeming hidden and distant. The change in `a()` of `p1` results in `b()` to match the if statement on line 14 and return 0 instead of doing an addition of `x` and `y` in `p2` (line 15). This further propagates to the `client` where `b()` is called. Similarly, the change in `v()` flips the condition to `false` instead of `true` in `p2` which result in skipping `y+2` at line 12. These two code changes illustrate how the client behavior or the execution flow is not honored after updating to a newer version.

Unlike breaking API contracts, semantic changes are not inherently bad: the refactoring of `z()` in `p2` introduces a new execution path (e.g., new behavior) to `make_false` which continues to return `false` after the change. Source code changes that preserve the same behavior before and after an update are semantic backward compatible changes. Deciding semantic backward compatibility is also a contextual problem: Given another client, `client2` that use the same dependency `p2` as `client` but don't call `b()` and `z()` (line 4). The same update we illustrate for `client` is semantic backward compatible for `client2` as it functions the same way before and after the update.

Following the observations in Listing 1, we denote a semantic backward compatible update or *safe update* as the following: We denote $Lib_1, Lib_2 \in Library$ as two versions of the same library and a client C with dependency tree as $T_C = (V, E)$ where V is a set of resolved versioned libraries used by C, and E is the directed dependence between them. Let PDG_{TC} represent a sound *program-dependence graph* (Ferrante et al., 1987) of T_C connecting data and control dependencies between program statements in both client and dependency code. The transition $[Lib_1 \rightarrow Lib_2]_C$ represents replacing Lib_1 with Lib_2 in client C. We arrive at the following definition of a safe update:

Definition 2.1. Given that $Lib_1 \in T_C$ and a request by a package manager to perform $[Lib_1 \rightarrow Lib_2]_C$, let $D = Lib_1 \setminus Lib_2$ be a source code diff mapping between Lib_1 and Lib_2, and function $f : D \rightarrow Y$ determine semantic compatibility for diff $d_i \in D$ in client C where $Y \in \{true, false\}$, an automatic update (or safe update) can only be made if and only if $\forall d_i \in D, f(d_1) \wedge f(d_2) \ldots \wedge f(d_n) = true$ where i varies from 1 to n and n is the cardinality of set D.

3. Research questions

The goal of this paper is to understand how reliable test suites are as a means to evaluate the compatibility of updated library versions in projects. To that end, we study a large number of test suites from Maven-based Java projects that depend on external libraries.

Bogart et al. (2016) report that developers create strategies to select high-quality libraries based on signals such as active contributors, project history, and personal trust in project maintainers to reduce the exposure of unwanted changes. Thus, in our first research question, we investigate whether testing of third-party libraries is prevalent and a strategy to minimize the risk of breaking changes:

RQ1: Do test suites cover the uses of third-party libraries in projects?

A qualitative study by Mirhosseini and Parnin (2017) suggests that developers have trust issues with automated updates and perceive tests as unreliable. A compelling complement to evaluate the effect of dependency changes is the use of change impact analysis. We set to measure how capable both test suites and change impact analysis can catch simple semantic faults in both direct and indirect uses of third-party libraries:

RQ2: How effective are project test suites and change impact analysis in detecting semantic changes in third-party library updates?

While static analysis can yield higher coverage, it is also more prone to false positives by classifying safe updates as unsafe. To understand the strengths and weaknesses of static analysis in a practical environment, we ask:

RQ3: How useful is static analysis in complementing tests for compatibility checking of new library versions?

We extract a set of real-world update cases from pull requests generated by the popular service Dependabot and manually investigate the correctness of each pull request. Then, we analyze each pull request using change impact analysis to compare the results with the test suite and our ground truth.

4. Research method

We follow the study design depicted in Fig. 2 to evaluate the reliability of test suites for automated dependency updating and the potential of using static analysis. First, we select Java repositories with high-quality assurance badges and at-least one test class from GitHub [1]. Then, we build each repository to infer a complete dependency tree of the project along with its

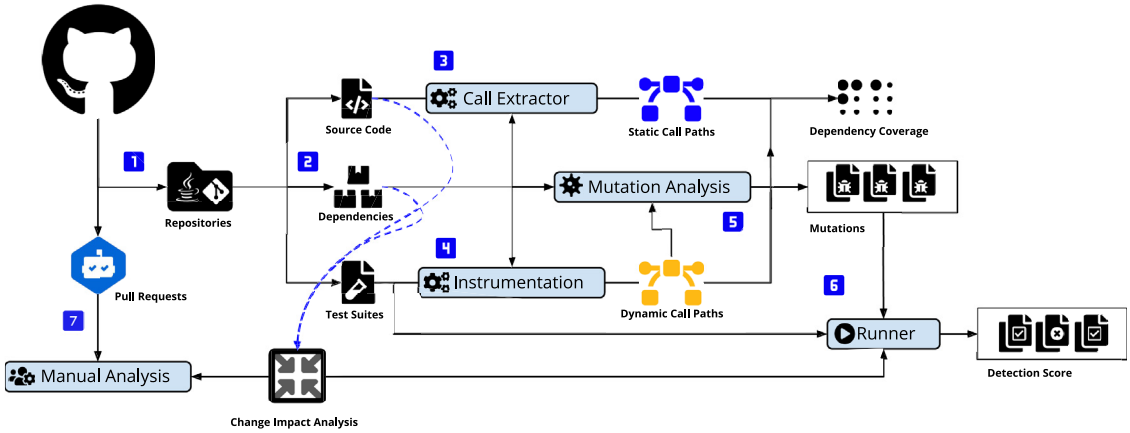

Fig. 2. Overview of our study infrastructure.

Table 1
Mutation operators (based on Papadakis et al., 2019).

Names	Description	Example
ABS	Absolute Value Insertion	$v \longmapsto \texttt{abs}(v) \mid -\texttt{abs}(v) \mid 0$
AOR	Arithmetic Operator Replacement	$x \text{ op } y \longmapsto x + y \mid x \,\%\, y \mid x \,/\, y$
LCR	Logical Connector Replacement	$x \text{ op } y \longmapsto x \mid\mid y \mid x \,\&\&\, y \mid x \,\hat{}\, y$
ROR	Relational Operator Replacement	$x \text{ op } y \longmapsto x > y \mid x != y \mid x >= y$
UOI	Unary Operator Insertion	$v \longmapsto v{+}{+}, {+}{+}v, !v$

source- and test classes in 2. Second, we feed the source classes together with the dependencies of a project to the call extractor and statically extract all its direct and indirect uses of third-party libraries 3. Third, we use instrumentation to learn all invocations from a project to its dependencies via its test suite 4. Then, we use the information from the previous step to calculate the dependency coverage of a project. Fourth, we generate mutations of dependencies by inserting simple faults (See Table 1) in dependency functions executed by tests. Here, we use dynamic call paths (from 4) to identify such functions 5. We can then run both the test suite and the change impact analysis to measure the detection score 6. Finally, we harvest Dependabot pull requests in a real-time fashion and then manually evaluate how both test suites and change impact analysis perform in practice 7.

4.1. Identifying usages of third-party libraries

We refer to the use of third-party libraries as using functionality from externally-developed libraries in software projects. Specifically, we focus on functionality exposed as functions in libraries as they are among the most widespread forms to achieve code reuse. Thus, we consider a function call from a project to a library dependency as third-party library use. As projects depend on an ordered tree of library dependencies, there are both implicit and explicit third-party uses. An explicit use is a direct function call between a project and one of its declared libraries. On the other hand, implicit use is when a function in a project transitively calls underlying libraries in a dependency tree. Given the following example scenario: project `A` depends on library `B`, and library `B` depends on library `C`. If there is a function call path between a function `a()` in `A` to a function `c()` in `C` via called functions in `B`, project `A` is implicitly using functionality in library `C`.

To identify explicit use of third-party libraries, we statically extract all function calls to functions that are neither part of the project under analysis or the Java standard library. By deduction, all such method invocations represent calls to third-party libraries. For implicit use of third-party libraries, we statically derive call graphs capturing call paths between a project and its dependency tree, similar to Ponta et al. (2018). Finally, we prune function and call sequences belonging to the Java standard library to derive a graph representing interactions between a project and its transitive dependencies.

To measure dependency coverage in a project in **RQ1**, we use instrumentation to record all project invocations to third-party libraries during test suite execution. Using the recorded set, we calculate the proportion of statically inferred functions covered by the test recorded set of function as dependency coverage (*Recorded functions* $\subset$ *Declared functions*):

$$Cov_{dep} = \frac{Recorded\ functions}{Declared\ functions}$$

Effectively, dependency coverage is function coverage (Myers et al., 2011), but only restricted to dependency calls.

4.2. Heuristics for static impact analysis

The central task of automated dependency updating is to facilitate the continuous integration of new compatible library versions with minimal developer intervention. Unlike static analysis that may contain false warnings (Beller et al., 2016), automated updating suffers instead from false negatives. A faulty update has a potentially high maintenance penalty if merged into the project and could cascade into breaking the build of externally depending projects.

As a step towards reducing false negatives, we are investigating change impact analysis as a means to potentially reduce coverage gaps where tests are not able to reach in dependencies. Change impact analysis estimates the reach and fraction of affected execution paths in a program given a set of code changes (Arnold, 1996). While there are advancements towards inference of semantic changes in static analysis such as data flow analysis with equivalence relations (Gyori et al., 2017) and mining techniques (Nguyen et al., 2019), precise static interpretation of semantic changes such as faulty updates is an undecidable problem (Emanuelsson and Nilsson, 2008). Moreover, most of these

techniques only analyze method bodies, and thus not practical for inter-procedural analysis of projects and their dependencies.

Without possibilities to precisely determine if an update is faulty or not, we approximate a faulty update (or semantic change) as a change to the execution flow of a project. We use control flow graphs (CFGs) (Allen, 1970) to represent all possible execution paths of functions. There are two types of statements in CFG terminology that affect the execution flow of a program, namely *control* and *write* statements (Zeller, 2009). A change to a write statement can affect the program state (i.e., assign a value to a variable). A change to a control statement changes the program counter (i.e., determine which statement to be executed next). By reading the program state, changes to the two other statements passively impacts *read* statements. Thus, we derive the following heuristics to classify unsafe updates:

Definition 4.1. Given a diff mapping $D = Lib_1 \setminus Lib_2$ between code entities in Lib_1 and Lib_2, we consider a code change as not *semantic preserving* if and only if $d_i \in D$ has a source location with a reachable control flow path to client C and maps to the following potential actions in a CFG:

1. d_i translates to change in the expression of *write* or *read* statements (data-flow change)
2. d_i translates to moving a statement from position x to y (control-flow change)
3. d_i translates to removing or expanding with new control flow paths (control-flow change)
4. d_i translates to changes in branch conditions (control-flow change)

The definition is an over-approximation; code changes such as code refactorings would be classified as an unsafe update if and only if affected functions are reachable. As services such as Dependabot present only the outcome of test results and a changelog between the old and new version of a library, change impact analysis instead precisely pinpoint affected execution paths in an update. Such information help project maintainers prioritize testing efforts or determine the potential risk of the update.

4.3. Creating unsafe updates in project dependencies

For seamless integration, it is important for automated dependency updating services to detect incompatibilities that arise when updating a library dependency. By using mutation analysis to seed artificial faults in all uses of third-party libraries in a project, we can derive an adequacy test of detecting incompatibilities in automated dependency updates. We first dynamically extract a set of called third-party functions in a project and then apply mutation operators defined in Table 1 to construct a set of artificial updates that are false negatives. As static analysis can over-approximate execution paths (i.e., risk creating false-positive cases), we resort to dynamic analysis to ensure mutations of truly invoked functions. For the selection of mutation operators, we choose operators common in mutation testing studies (Just et al., 2014; Papadakis et al., 2019) that focus on simple logical flaws and exclude mutation operators with a limited effect such as deleting statements (Just et al., 2014).

In comparison to using actual update cases, the mutation setup provides a systematic way to introduce simple faults in **all** uses of third-party libraries in a project to measure the effectiveness of detecting faulty updates. Manually curating false-negative cases of dependency updates limits to specific project-library pairs and may not generalize to other projects that use the same library. Moreover, finding such pairs for all libraries in a project to create an overall assessment may not be possible in most projects.

For **RQ2**, we denote *mutation detection score for dependencies* (an adaption of *mutation score* (Just, 2014)) as a tool's ability to detect a mutated reachable dependency function as (mutants):

$$\textit{Detection Score} = \frac{\textit{Detected mutants}}{\textit{All mutants}}$$

4.4. Manual analysis of pull requests

As the artificially created updates address only false negatives, we also need to understand how static analysis performs in practice. Thus, we manually analyze the applicability of static analysis using pull requests through a lightweight code review. Due to the absence of established ground truth or a benchmark, we resort to manually creating a ground truth of libraries under update. As understanding the use context of a project-library is challenging, we also, attempt to corroborate our findings by posting our assessment as pull request comments. Below, we define our setup for the manual analysis:

Selection criteria. We select pull requests generated from the popular service Dependabot on GitHub that supports automated updates of Java projects using the Maven-build system. To select significant and high impactful projects and increase the chance for a response by a project maintainer, we harvest newly created pull requests using GHTorrent's event stream (Gousios and Spinellis, 2012) and adopt the following filter criteria: (1) *high stargazer, watchers or forks count* indicate popularity, (2) *no passive users* indicate projects that assign reviewers and frequently merge Dependabot-pull requests, (3) *dependency type* indicates that we only consider Maven compile and runtime dependencies, and (4) *project buildability* indicates that we can compile the project out of the box.

Code review protocol. After a pull request meets the selection criteria, we first inspect the diff in the pull request to identify the old and the new version number of the library under update. Then, we download the source jar of the old and new version from Maven Central and use a diffing tool to localize the set of changes. By reviewing the change location, consulting the changelog, inspecting the tests of the library, we classify the nature of a change as refactoring, structural (i.e., breaking change), or behavioral (i.e., semantic change). Next, we check out the project at the commit described in the pull request and manually localize uses of the library by first performing keyword search of import statements leading to the library under update. Then, we track the data- and control-flow of imported items (e.g., object instantiations, function invocations, and interface implementations) to map out how the project uses the library under update. If the library under update is a transitive dependency, we first trace how the project uses its direct dependency and then how the used subset of the direct dependency uses the transitive dependency. After mapping out uses of the library under analysis, we can then establish whether a project directly or indirectly uses any of the changed classes and function signatures identified in the diff and whether those changes make the update safe or not. If the changes do not alter the logic (e.g., refactorings) of the project, we consider the update safe. Refactorings are in some cases highly contextual and can yield different outcomes as exemplified in the following: the changed function `foo(x)` adds a new IF-statement with the condition `x > 50` that breaks the original functionality. Project `A` uses `foo(x)` indirectly, and through the manual analysis (including inspection of its tests), we can establish that the threshold is $x < 20$ in all cases, and thus the update is safe to make. On the other hand, project `B` has a public function `bar(x)` that passes `x` in a function call to `foo(x)`. Here, we cannot assume anything around `x` as users

of B could call `bar(x)` with any x. In this case, we consider the update unsafe.

After manually evaluating pull requests, we classify them using one of the three categories:

- *Safe*: the update is safe to perform and will not negatively impact the functionality of the project.
- *Unsafe*: the update is risky and could lead to potential unexpected runtime changes.
- *Unused*: the update of an unused dependency (i.e., it is only declared in the project but not used).

Based on the outcome of the update tooling, we compare it with the classification above and consider the following:

- False Negative (FN) when classifying an unsafe update as safe.
- False Positive (FP) when classifying a safe update as unsafe or falsely updating an unused dependency.
- True Positive (TP) when both our manual classification and update tooling has the same conclusion.
- True Negative (TN) when not creating an update for an unused dependency.

4.5. Dataset construction

We sample 1823 repositories from GitHub that have Java as the primary language, Maven as the primary build system, and have at-least a high-quality assurance badge (i.e., Travis CI, CodeClimate, coveralls, and CodeCov) as a signal for having tests (Trockman et al., 2018). Services such as Dependabot can update dependencies in projects as long as there is a valid `pom.xml` file. Next, we build and then dry-run projects on both the instrumentation and mutation pipeline to eliminate incompatible projects. In total, there are 818 repositories that compile to Java 8 bytecode and have at least one compiled test class. Out of the 818 built projects, 521 projects successfully run the instrumentation pipeline, and a subset of 262 projects are compatible with the mutation pipeline. The number of projects in the mutation pipeline is nearly double the ratio of a recent previous study (Zhang et al., 2018). Table 2 presents descriptive statistics on four aggregated variables for projects belonging to the instrumentation pipeline. The median number of declared methods is 210 (mean: 668) with a heavily positive skewed distribution. 75% of all projects in our sample cluster around 588 or less declared methods with 36 projects having more than 1400 methods. The largest project is `oracle/oci-java-sdk` with 22,264 methods. As per Section 4.1, we measure test coverage of all function calls made in a project. We can observe that the test coverage is generally high: half of the projects have coverage of 67% or more. For the number of dependencies, we can observe that the distribution does not drastically change: the median changes from 7 to 16, indicating a small expansion of transitive dependencies. Overall, our dataset represents mid-sized projects that use a significant number of dependencies with varying test coverage.

4.6. Implementation

We discuss the implementation of Uppdatera, a tooling for performing change impact analysis of library dependencies in Maven, and our pipeline to run our experiments. We have open-sourced the tooling and docker images for automation and reproducibility of our study (see Section 6.3).

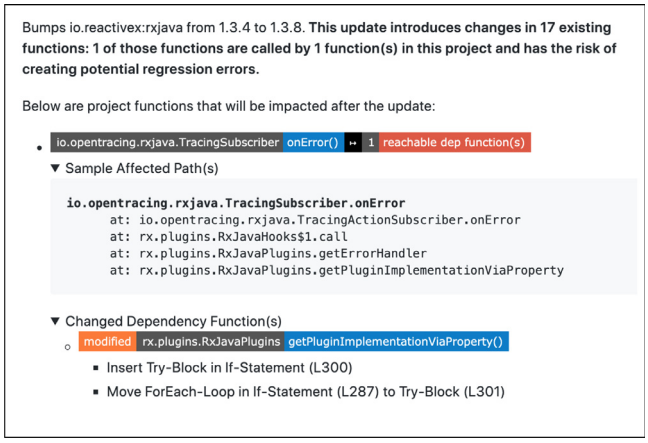

Fig. 3. Example of updating `rxjava` from 1.3.4 to 1.3.8 in the project `opentracing-contrib/java-rxjava`.

4.6.1. Uppdatera

Given a request to update a dependency to a new version in a `pom.xml` file, Uppdatera first performs AST differencing of the current and new version of the dependency to identify a list of functions with potential behavioral changes using `SpoonLabs/GumTree` (Falleri et al., 2014). Then, Uppdatera computes a call graph inferring all control-flow paths between client and dependency functions following Ponta et al. (2018)'s approach for call graph construction (using WALA). Finally, Uppdatera performs a reachability analysis using the list of possible behavioral changes on the call graph to find reachable paths to the client code. Fig. 3 demonstrates an example of using Uppdatera for updating the library `io.reactivex:rxjava` from version 1.3.4 to 1.3.8 in `opentracing-contrib/java-rxjava`. The report features a call stack to the changed function along with a set of AST diffs. In this particular case, the `onError()` function in the class `TracingSubscriber` transitively calls `getPluginImplementationViaProperty()` in the dependency class `RxJavaPlugins`. The addition of a `try-catch` block in the function takes care of unhandled exceptions which may have been handled by clients in previous versions (i.e., potential regression change)

In the following paragraphs, we motivate our implementation choices for a change impact analysis tool designated for updating library dependencies.

Diffing. Uppdatera performs source code differencing at the abstract syntax tree (AST) level of both the current and the new version of a dependency to identify functions with code changes. AST differencing algorithms (Fluri et al., 2007; Falleri et al., 2014) produce fine-grained and accurate information about the type and structure of source code changes. Following Definition 4.1, we capture AST transformations at the statement level and map the following as regression changes:

- Any method-level *move* operation mirrors moving a statement from line *x* to *y*.
- *deletion*, *update* or *insertion* of *Expression* ASTs mirrors data-flow changes.
- *deletion*, *update* or *insertion* of control struct ASTs such as *IF*, *While*, *FOR* mirrors control-flow changes.
- *deletion*, *update* or *insertion* of *Call-Expression* ASTs represents changes mirrors control-flow changes.

As an alternative to AST differencing, we could consider bytecode differencing. Bytecode (e.g., LLVM's IR or JVM code) differencing compute edit scripts at the instruction level. Although

Table 2
Descriptive Statistics for 521 GitHub projects (each variable aggregated per project).

Variable	Unit	$Q_{0.05}$	Mean	Median	$Q_{0.95}$	Histogram
Project methods	Count	20	668	210.5	2320.5	
Test coverage (function calls)	%	7	72	64	97	
Direct dependencies	Count	1	10	7	31	
Transitive dependencies	Count	1	31	16	105	

this technique offers a fine-grained and a compelling alternative to AST differencing, instruction-level changes can be difficult to understand for developers not familiar with low-level details.

Call graph construction. Uppdatera constructs a call graph capturing inter-procedural control-flow paths between client and dependency functions. Each node in the call graph represents a fully resolved function identifier and should be identical to the identifiers in the changeset of the **Diffing** phase.

We advocate the use of call graph algorithms that are both *soundy* (Livshits et al., 2015) and scalable for analyzing projects in the wild as a general guideline. The call graph algorithm should support and resolve as many language features as possible. Limited support of language features could potentially leave gaps in the coverage of projects making use of unsupported features. Similar to static analyses of security applications, achieving high recall is more crucial than precision to avoid recommending faulty updates.

As recent studies (Kikas et al., 2017; Decan et al., 2018b) suggest that irrespective of the OPR, the majority of packages have a small number of direct dependencies, but a high and growing number of transitive dependencies. For example, 50% of all packages in Crates.io have a dependency tree depth of at least 6 (Decan et al., 2018b). Therefore, performing static analysis at the boundary of a project and its dependency tree can become computationally expensive and impractical in DevOps environments. Moreover, as Uppdatera can expect to analyze any compatible project in the wild, the algorithm should be scalable to cater large projects and cheap to construct to cut down computation time.

Finally, a potential trade-off of using call graphs instead of CFGs is the loss of analysis precision due to the absence of data-flow paths in the graph. However, taking into account program features such as aliases, arrays, structs, and class objects in dataflow analysis adds additional complexity and scalability problems when moving the analysis boundary to include project dependencies. Supporting such analysis adds extra precision but may not yield extra actionability.

Reachability analysis. For each changed function identified in the **Diffing** phase, Uppdatera performs a reachability analysis on the call graph to detect paths connecting changed dependency functions to functions in the analyzed project. If Uppdatera finds such paths, it marks the update as potentially *unsafe*. If no such paths are found, Uppdatera marks it as a potential *safe* update and recommends the update to the package manager. Finally, Uppdatera also reports the impacted paths between dependencies and project functions, to inform developers of the program paths that need to be inspected in response to an update in a dependency.

4.6.2. Experimental pipeline

To implement our methodology, we first develop a call extractor that records complete call sequences between a project and its library dependencies. The implementation builds on instrumenting library classes using ASM (Bruneton, 2011) and the Maven Dependency Plugin. To infer function calls to libraries from a project (**RQ1**), we use ASM to statically extract call sites for direct dependencies. We generate call graphs using WALA (IBM Research, 2006) configured for the CHA algorithm for extracting calls to transitive dependencies. Following Reif et al. (2019)'s comprehensive benchmark of call graph algorithms for Java, we find that the CHA algorithm supports the most language features and has a lower runtime in comparison to more precise points-to analysis algorithms such as 0-1-CFA or N-CFA.

For **RQ2**, we implement the update emulation pipeline (i.e. mutation analysis) on top of PITest (Coles et al., 2016), a popular in-memory-based mutation testing framework that works with the popular test runners JUnit and TestNG by limiting mutations to library functions identified from the call extractor. We exclude the use of experimental mutation operators that cannot guarantee non-equivalent mutations. For each mutated class, we use Procyon (Strobel, 2016) to decompile into a source file for AST diffing in the case of Uppdatera.

5. Results

Here, we report the results of our research questions.

5.1. RQ1: Dependency coverage

Fig. 4 presents a violin plot of dependency coverage on the left-hand side, and dependency coverage including transitive dependencies on the right-hand side. Overall, 13% (67/521) projects have less than 10% coverage, suggesting at large that a majority of projects have some tests exercising at least one dependency use. We observe that the median coverage is 58% (mean: 55%): half of the GitHub projects miss coverage of more or at least 42% of all dependency function calls. In practice, there is a risk that automated dependency updating may not have tests that exercise changes in dependencies.

The right-hand side of Fig. 4 shows the dependency coverage taking into account reachable paths to transitive dependencies in projects. The distribution has a bimodal shape with two peaks at, 9%, and at 52%, suggesting two classes of projects. In the first class, half of the projects have a median dependency coverage of 21% (mean: 26%), indicating that project test suites at large do not exercise dependencies in depth. This is not surprising: an ergonomic factor of third-party libraries is that they are well-tested and should in principle not need extra tests (Cox, 2019). In the second class, we can observe that projects have tests that exercise dependencies in-depth, suggesting the presence of projects with adequate test suites. As mentioned in Section 4.1, these results are indicative as we compare against statically inferred call paths, which, being over-approximating, may not be representative of actual calls.

Findings from RQ1: *Half of the 521 projects exercise less than 60% of all direct dependency calls from their tests; this drops to 20% if paths to transitive dependencies are considered.*

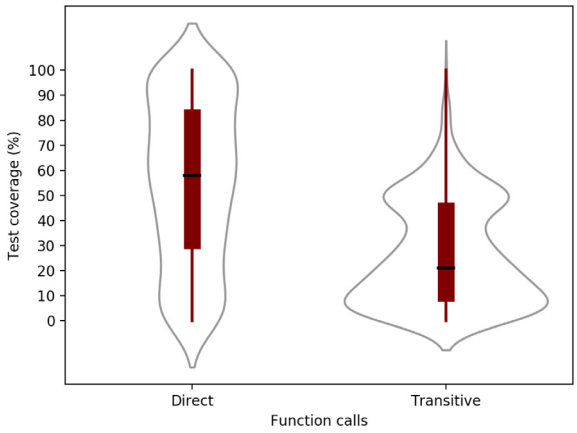

Fig. 4. Test coverage of dependencies.

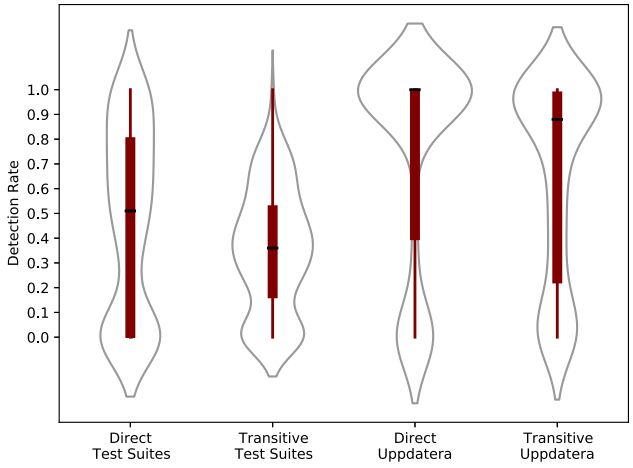

Fig. 5. Mutation detection score.

5.2. RQ2: Detecting simple faults in dependencies

Our benchmark generated in total 1,122,420 artificial updates for 311 Maven modules belonging to 262 GitHub projects. Fig. 5 shows a violin plot of the mutation detection score for both direct and transitive dependencies, split by project test suites on the left-hand side and Uppdatera on the right-hand side. The median detection rate score is 51% (mean: 47%) for direct dependencies and 36% (mean: 35%) for transitive dependencies. We can observe that 25% of the projects have a high test suite effectiveness greater or equal to 80% for direct dependencies. When looking at transitive dependencies, the median of direct dependencies and the third-quantile of transitive dependencies are similar, showing that only one-fourth of the test suites remain effective in detecting faults in transitive dependencies. Moreover, we can also see more dispersion in effectiveness among direct dependencies than transitive dependencies, half of the projects have a detection score ranging between 16 to 54% for transitive dependencies. Overall, the results indicate that tests are effective for a limited number of cases and dependencies. At large, however, a small minority of projects have test suites that can comprehensively detect faulty updates.

On the other hand, Uppdatera, has a median detection score of 97% (mean: 74%) for direct dependencies and 88% (mean: 64%) for transitive dependencies. Generally, we see that static analysis is highly effective in detecting simple faults with a slightly decreased effectiveness for transitive dependencies. Half of the projects with a low detection score (<50%) using tests now have detection score greater than 80%. In the lower half of the median for both direct and transitive dependencies, we see large variations between the projects. As change impact analysis is largely a generic technique, we manually investigate why Uppdatera was unable to detect changes in 76 modules having a low detection score of less than or equal to 39% and 22% for direct and transitive dependencies, respectively. We perform a manual investigation using the following protocol: (1) back-track from the dynamic call trace to test suite, (2) identity potential tests cases that invoke the path in the call trace, and (3) investigate both the test case setup and source code in-depth to understand how Uppdatera could miss the regression change in the update.

In total, we identified four potential reasons for Uppdatera to miss faulty updates: 29 cases involving code generation, 26 cases involving class loading, 19 cases involving instrumentation, and 2 cases of instantiations of generic methods. Dynamic class loading along with code generation makes use of Java's Reflection API such as `Class.forName(DynClass)`. A majority of the inspected cases stem from libraries such a FasterXML Jackson-databind, Jersey REST framework, Spring framework, JAI ImageIO, Hibernate Validator, and Google Guice. Reflection is useful in cases such as the creation of data bindings (jackson-databind), data validation (hibernate or guice) or generation of HTTP endpoints from annotated user methods (jersey or spring framework). Resolving cases involving reflection is a known limitation of static analysis (Reif et al., 2019).

Although we do not instrument JUnit and Maven (which we use to power our setup), projects can bypass our exclusion filter by putting those libraries under a different namespace, a practice known as *class shading*. We identify several instances of bypassing the filter, an effect we cannot easily control. Finally, in two cases, generic methods defined in user projects were only instantiated in tests but not in the project source code. Generally, call graph generators do not resolve generic methods unless there is a concrete instantiation of it.

> **Findings from RQ2:** *Project tests are effective in a limited number of cases but not at large.* Uppdatera *can detect twice as many faulty artificial updates as opposed to project test suites. Libraries making use of Java's reflection API could affect its applicability.*

5.3. RQ3: Change impact analysis in practice

We conducted our online monitoring for two weeks between 13–27 Apr 2020 evaluating in total 22 Dependabot pull requests. On average, we harvested around 350 pull requests per day between Mondays and Wednesdays, 150 pull requests per day between Thursday and Fridays, 50 pull requests per day on the weekends. While the number of pull requests may seem high, a majority of them were updates of Maven plugins or test dependencies, uncompilable, or superseding previous pull requests. Thus, we posted on average two pull requests per day taking anywhere between one to four hours to manually evaluate pull requests and post our findings as comments.

Table 3 presents the analyzed pull requests along with the update type, ground truth class (i.e., **Class** column), external confirmation (i.e., **Confirm** column), results from the tooling, and execution times (in minutes). In total, our ground truth consists of 15 pull requests where the update is safe (i.e., **S** class), three pull requests where the dependency under update is unused and only declared (i.e., **N** class), and four pull requests where the updates that are unsafe (i.e., **U** class). The test suites of the analyzed pull requests classified 15 update as true positives (TP), four update cases as false positives (FP), and three

Table 3
Results of running Uppdatera on 22 Dependabot pull requests.

Pull Request	Update Type	Class	Confirm	Test Suite	Uppdatera	Test Runtime	Uppdatera Runtime
spotify/dbeam#189	Patch	**S**	✓	FP	FP	3.11	2.31
airsonic/airsonic#1622	Minor	**S**	✓	TP	FP	77	7.5
bitrich-info/xchange-stream#570	Patch	**S**	✓	TP	FP	2.78	1.93
CROSSINGTUD/CryptoAnalysis#245	Major	**S**	✓	TP	FP	12	2.6
dbmdz/imageio-jnr#84	Patch	**S**	✓	TP	FP	–	0.7
dnsimple/dnsimple-java#23	Minor	**S**	✓	TP	TP	2	11
smallrye/smallrye-config#289	Patch	**S**	✓	TP	TP	1.1	0.6
dropwizard/metrics#1567	Patch	**S**	✓	TP	TP	2.6	8.73
s4u/pgpverify-maven-plugin#96	Minor	**S**	✓	TP	TP	4	1.2
JanusGraph/janusgraph#2094	Minor	**N**	✓	FP	TN	365	33
UniversalMediaServer/UniversalMediaServer#1989	Major	**U**	✓	FN	TP	11	8.3
premium-minds/pm-wicket-utils#71	Patch	**N**	✓	FP	TN	1.56	0.51
UniversalMediaServer/UniversalMediaServer#1987	Minor	**N**	✓	FP	TN	11	7.7
CSUC/wos-times-cited-service#36	Patch	**S**	✗	TP	FP	0.55	0.5
Grundlefleck/ASM-NonClassloadingExtensions#25	Major	**S**	✗	TP	FP	4	0.5
RohanNagar/lightning#211	Major	**U**	✗	TP	TP	2	7.8
zalando/riptide#932	Minor	**U**	✗	FN	TP	7.5	20.5
pinterest/secor#1273	Patch	**S**	✗	TP	TP	390	13.5
michael-simons/neo4j-migrations#60	Patch	**S**	✗	TP	TP	3.45	0.8
zaproxy/crawljax#115	Minor	**U**	✗	FN	TP	17.35	1.3
hub4j/github-api#793	Minor	**S**	✗	TP	TP	1.3	4
zalando/logbook#750	Patch	**S**	✗	TP	TP	6.1	18.38

update cases as false negatives (FN). Uppdatera classified 12 update cases as true positives (TP), seven cases as false positives (FP), three cases as true negatives (TN). There are 12 cases where the two techniques report differently as highlighted by the colors in Table 3. Most notable are false positives; Uppdatera incorrectly reports six updates (highlighted yellow in the table) as unsafe that test suites can detect as safe. In those cases, the heuristics failed to account for refactorings or falsely derived call paths due to dynamic dispatch. In four cases, Uppdatera could not detect that the changes were refactorings (i.e., semantic-preserving changes). One such example is `airsonic/airsonic#1622`, where a confirmed minor update of the Apache `commons-lang3` library involved refactoring array length and null checks into a new function. In the two remaining cases, all reachable call paths were over-approximations. The update of `org.eclipse.emf.common` in `CROSSINGTUD/CryptoAnalysis#245` included changes to List structures implementing methods of Java's List Interface (such as `addAll()`), resulting in unrelated interface calls being linked to it. This is a limitation of the CHA algorithm as it links interface calls to all available implementations. In the three confirmed **N**-cases (highlighted blue in the table) where tests would falsely pass the updates, Uppdatera correctly identified no use of the dependency under update in the projects. The project maintainers in two of the reported cases have started refactoring work to remove those identified dependencies.

Uppdatera was able to complement test suites in three false-negative cases (highlighted red in the table). In our confirmed case of an unsafe update, Uppdatera identified the Apache `commons-lang3` library to break the application logic in `UniversalMediaServer/UniversalMediaServer#1989` due to changes in calculating string edits using the Jaro–Winkler distance. Generally, we can observe that solely using static analysis may risk falsely classifying safe updates as unsafe. Finally, we also make a comparison of execution times between running tests and Uppdatera. The results reveals that Uppdatera has faster or comparable times in 16 out of 22 cases, suggesting that change impact analysis can be a viable option to complement tests in CI environments.

Findings from RQ3: *Semantically equivalent changes (refactorings) and over-approximated function calls are the main sources of false positives in* Uppdatera. *However,* Uppdatera *helped project maintainers identify risky updates and unused dependencies.*

6. Discussion

6.1. Evaluating library updates

Updating to a new version of a third-party library is not a trivial task, and for good reasons: interface refactorings induce additional maintenance burden and integrating untested behavior can jeopardize project stability. Services such as Dependabot advocate a modest update strategy focusing on project compatibility: only update if the tests pass with the new library version. Effectively, developer-written tests act as the first-line defense against library updates introducing regression changes.

A key insight in our work is that automated dependency updates are not reliable. Our results strongly suggest that existing developer-written tests lack specifications that exercise dependencies in depth. This finding is in line with the work by Mirhosseini and Parnin (2017), where developers report being suspicious of integrating automated updates due to fear of breakage. When selecting to adopt a third-party library, Bogart et al. report that developers look at aspects such as reputation, code quality standards and active maintenance to build up trust (Bogart et al., 2016). Perceived high-quality libraries can eliminate the need for extensive testing. In our case, we found evidence against this practice. The minor backward-compatible update of `org.apache.commons:commons-lang3`, a high-quality library, had changes that would break the application logic in one project if the pull request was merged in our manual analysis. In addition, the practice of testing third-party libraries is not common among popular testing books (Whittaker, 2002; Myers et al., 2011; Hetzel and Hetzel, 1988), very few research papers suggest testing of third-party libraries (Kropp et al., 1998; Mariani et al., 2007).

Directing testing efforts to dependencies would be a potential solution to the problem. Therefore, we recommend practitioners to use automated updating services cautiously and complement with tests for critical library dependencies. For tool creators in the domain, we argue for increased transparency in automated updating. With a small minority of projects having both coverage and tests capable of detecting simple regressions, pull requests could feature a confidence score on how well it is able to test new changes in a library under update. As a first step, tool creators can make use of our study setup to measure both coverage and quality of tests as an indication of confidence. A confidence score could also help reduce false negatives: if no tests are exercising a changed functionality of a dependency under update, Dependabot could avoid recommending it.

6.2. Strengths and weaknesses of static analysis

Without needing to maintain additional dependency-specific tests, static analysis can be effective in deterring updates with potential regression changes. For a large number of projects with limited test quality, change impact analysis can fill the gap where tests are unable to reach and would be a compelling option for tool creators to consider. For a minority of projects, however, we identify certain third-party libraries that impede the overall analysis accuracy. Libraries heavily relying on code generation such as the Spring framework makes use of the Java Reflection API that are known to be statically difficult to analyze (Antoniadis et al., 2020), could miss critical execution paths in projects that make use of them. Moreover, by linking interface calls to all its implementations, call graphs contain over-approximated call paths. We could observe non-existing interface calls from functions in the unused dependency to classes implementing the interface in the project during the manual analysis. As Ponta et al. (2018) approach base on building a call graph with the project and its dependencies together, we make preliminary observations that projects having library dependencies with several common interfaces between them are likely to have many unrelated function calls. Exploring improvements such as using type hints with data flow analysis could potentially eliminate such function calls. Overall, we argue that static analysis is a useful complement in use cases where tests lack coverage. By also revealing and presenting gaps and quality issues in test suites, static analysis can help developers in prioritizing testing efforts of dependencies.

6.3. Threats to validity

Sampling random projects from GitHub pose threats to our results: tests or dependencies in projects may not exercise production classes. To mitigate this risk, we configure our call extractor to only record call paths originating from the project source code. Call paths that do not traverse via project source code are excluded (e.g., test class directly calling a dependency).

The use of mutation analysis to emulate source code changes in dependency functions has several potential threats to validity. First, we acknowledge that the applied mutation operators do not substitute actual regression changes in library updates. Our objective is to exercise all uses of libraries in a project by injecting simple faults to uncover potential coverage gaps in updating tools. Using real-world cases for this purpose would be challenging and potentially adding hidden uncontrolled factors. Second, our ground truth in **RQ2** represents reachable call paths inferred from running project tests, making it a subset of all possible executions and is a limitation of the benchmark. A potential avenue to explore is the use of test generation techniques such as EvoSuite (Fraser and Arcuri, 2011) to discover new call paths. However, EvoSuite generates tests at the class level without considering its interaction with other classes or dependencies, generating artificial tests that may not represent valid use cases.

The false-positive rate in **RQ3** is indicative and not representative. Without domain knowledge of the interplay between a project and a dependency, the code reviews may state incorrect or incomplete information. To mitigate this risk, we post our code review assessment in the update for the project maintainer to react in case of incorrect analysis. Finally, for the reproducibility of our study, we have made the source code,[1] the experimental pipeline,[2] and our data publicly available (Hejderup and Gousios, 2021). Specifically, we include the examined projects, applied mutation changes, and their dynamic and static call graph.

[1] https://github.com/jhejderup/uppdatera.

[2] https://github.com/jhejderup/uppdatera-pipeline.

7. Related work

Updating library dependencies in projects. To assist developers with updating dependencies in projects, researchers have studied practices around updating dependencies (Kula et al., 2018; Mirhosseini and Parnin, 2017; Bogart et al., 2016; Dietrich et al., 2019, 2014; Raemaekers et al., 2017) and proposed tools leveraging both static- and dynamic analysis (Foo et al., 2018b; Mezzetti et al., 2018; Møller and Torp, 2019). Kula et al. (2018) empirical study of 2700 library dependencies in 4600 Java project found that 81.5% remain outdated, even with security problems. The study found that factors such as uncertainties around estimating refactoring efforts and other task priorities as reasons for developers to not update dependencies. To address the update fatigue for developers, automated dependency updaters such as Dependabot and `greenkeeper.io` actively reminds and suggests dependency updates to developers through the use of pull requests. A study by Mirhosseini and Parnin (2017) found that pull requests encourage developers to update dependencies more frequently but the frequency of updates and lack of convincing arguments defer them from updating. On similar lines, the work of Bogart et al. (2016) also suggests that developers perceive the use of monitoring tools to have a high signal-to-noise ratio than giving actionable insights. Finally, the empirical work of Dietrich et al. (2014) suggests that 75% of emulated library updates in the Qualitas dataset has breaking changes. However, only a few updates resulted in an error, motivating the need for contextual analysis.

Recently researchers have started to explore the use of static- and dynamic analysis to identify library updates with breaking changes, saving developers time, and review efforts of library updates. `NoRegrets` (Mezzetti et al., 2018; Møller and Torp, 2019) is a tool that detects breaking changes in test suites of dependent npm packages before releasing an update of the library. Although helpful in minimizing the chances of breaking changes for clients, the identified subset of clients may not be representative of other clients. Similarly, Foo et al. (2018b) describes a static approach using simple diffing and querying Veracode's SGL (Foo et al., 2018a) graph to find clients affected by breaking changes. In contrast to this approach, Uppdatera analyzes at the project level (e.g., does not search for affected clients), targets diff with data- and control flow changes (i.e., not only interface changes), and includes a benchmark to compare updating tools.

Change impact analysis. Change Impact analysis is a widely studied problem in program analysis research (Li et al., 2013; Lehnert, 2011). Propagation of changes in package repositories have become an important research area in light of incidents such as the *left-pad incident*, and recent moves to emulate these problems on package-based networks (Abdalkareem et al., 2017; Kikas et al., 2017). Several techniques (Ryder and Tip, 2001; Badri et al., 2005; Ren et al., 2004; German et al., 2009; Li et al., 2012) use call graphs as an intermediate representation for change impact analysis. Alternative techniques to call graphs are static and dynamic slicing (Tip, 1994; Arnold, 1996), profiling (Law and Rothermel, 2003; Orso et al., 2003) and execution traces (Orso et al., 2004). Due to cost-precision trade-offs, several proposed approaches use a combination of these techniques. One such example is Alimadadi et al. (2015)'s work on Tochal, that leverages both runtime data and call graphs to more accurately represent changes to dynamic features such as the DOM. For a comprehensive overview of impact analysis techniques and change estimations, we refer the reader to Li et al. (2013)'s survey on code-based change impact analysis techniques

An application of change impact analysis is regression test selection techniques (Yoo and Harman, 2012) (RTS) such as class-based STARTS (Shi et al., 2019; Legunsen et al., 2017) and probabilistic test selection (Machalica et al., 2019) that find relevant

tests for evaluating new code changes. We found in our evaluation that test suites have limited coverage of dependencies, thus RTS may not be able to find tests relevant for changes in dependencies or have enough test data to build a prediction model for average GitHub projects. Finally, Danglot et al. (2020) and Da Silva et al. (2020) investigate the use of search-based methods such as test amplification and automated test generation for detecting semantically conflicting changes. Although search-based methods are effective in reducing false positives and to some degree eliminating false negatives present in static analysis, they are limiting for integration test scenarios such as automated dependency updating. Da Silva et al. (2020) found that automated test generation such as EvoSuite (Fraser and Arcuri, 2011) have difficulties in generating effective tests for complex objects with internal or external dependencies.

8. Conclusions and future work

In this paper, we empirically investigate the reliability of test suites for automating dependency updates. With an increasing number of developers relying on services that automate updates of dependencies, our goal was to uncover to what degree project tests exercise utilized functionality in library dependencies, how effective they are in catching simple regressions, and their performance in practice. As recent research highlights the need for conservative techniques, we explored the use of change impact analysis to reduce false negatives.

Our findings show that half of 521 well-tested projects with tests cover less than 60% of their function calls to direct dependencies. The coverage drops to 21% when considering call paths to transitive dependencies. By artificially injecting simple faults in library dependencies to 262 projects, we observe that one-forth of the projects can detect 80% or more faults in functions of direct libraries. When considering transitive dependencies, the number of projects drop to one-eighth. Conversely change impact analysis, can detect 80% of potentially breaking in changes in both direct and transitive dependencies, two times more than using test suites. Although change impact analysis is a promising direction to flag faulty updates, we also manually investigate whether it can complement tests in 22 Dependabot pull requests. Our results show that change impact analysis could avoid unsafe updates in three cases where tests failed and spotted unused libraries in three cases. However, there are more false positives with change impact analysis as it is more imprecise than tests.

Our findings suggest that developers that are making use of automated dependency updating need to be aware of the risks with using project tests for compatibility checking. Without coverage or adequate tests for all usages of library dependencies, updates can silently introduce unintended functionality over time. As services such as Dependabot do not advertise risks involved with updating dependencies, tool creators could introduce reliability measurements such as scoring test suites in pull requests. As we investigate the use of change impact analysis, we argue that tool creators should explore combining dynamic and static analysis to derive verification techniques that do not strongly depend on users' test suites.

In future work, we aim to establish best practices for updating third-party libraries. As a first step, we aim to understand whether developers direct testing efforts towards dependencies and uncover the strategies they use. Moreover, we also intend to explore hybrid workflows through data-driven methods for efficient update checking by combining dynamic and static analysis.

CRediT authorship contribution statement

Joseph Hejderup: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft, Writing – review & editing, Visualization, Project administration. **Georgios Gousios:** Conceptualization, Methodology, Validation, Writing – review & editing, Supervision, Funding acquisition.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

We thank Moritz, Xunhui, Mauricio, Ayushi, and Arie for reviewing drafts of this paper. The work in this paper was partially funded by NWO, The Netherlands grant 628.008.001 (CodeFeedr) and H2020 grant 825328 (FASTEN).

References

Abdalkareem, R., Nourry, O., Wehaibi, S., Mujahid, S., Shihab, E., 2017. Why do developers use trivial packages? an empirical case study on npm. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, pp. 385–395.

Agrawal, H., Horgan, J.R., London, S., Wong, W.E., 1995. Fault localization using execution slices and dataflow tests. In: Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95. IEEE, pp. 143–151.

Alimadadi, S., Mesbah, A., Pattabiraman, K., 2015. Hybrid dom-sensitive change impact analysis for javascript. In: LIPIcs-Leibniz International Proceedings in Informatics. 37, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Allen, F.E., 1970. Control flow analysis. In: ACM Sigplan Notices. 5, (7), ACM, pp. 1–19.

Anderson, B., 2018. Test the downstream impact of rust crate changes before publishing. https://github.com/brson/cargo-crusader (Accessed on 09 November 2018).

Antoniadis, A., Filippakis, N., Krishnan, P., Ramesh, R., Allen, N., Smaragdakis, Y., 2020. Static analysis of java enterprise applications: Frameworks and caches, the elephants in the room. In: Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.

Arnold, R.S., 1996. Software Change Impact Analysis. IEEE Computer Society Press.

Badri, L., Badri, M., St-Yves, D., 2005. Supporting predictive change impact analysis: a control call graph based technique. In: Software Engineering Conference, 2005. APSEC'05. 12th Asia-Pacific. IEEE, pp. 9–pp.

Bahmutov, G., 2014. Do not break dependant modules. https://glebbahmutov.com/blog/do-not-break-dependant-modules/ (Accessed on 09/11/2018).

Beller, M., Bholanath, R., McIntosh, S., Zaidman, A., 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. In: 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER). 1, IEEE, pp. 470–481.

Bogart, C., Kästner, C., Herbsleb, J., Thung, F., 2016. 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. ACM, pp. 109–120.

Brito, A., Xavier, L., Hora, A., Valente, M.T., 2018. Apidiff: Detecting API breaking changes. In: 2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 507–511.

Bruneton, E., 2011. Asm 4.0 a java bytecode engineering library. https://asm.ow2.io/ (Accessed on 18 April 2019).

Cleve, H., Zeller, A., 2005. Locating causes of program failures. In: Proceedings of the 27th International Conference on Software Engineering. ACM, pp. 342–351.

Coles, H., Laurent, T., Henard, C., Papadakis, M., Ventresque, A., 2016. Pit: a practical mutation testing tool for java. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. ACM, pp. 449–452.

Cox, R., 2019. Surviving software dependencies. Commun. ACM 62 (9), 36–43.

Cox, J., Bouwers, E., Van Eekelen, M., Visser, J., 2015. Measuring dependency freshness in software systems. In: 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering, vol. 2. IEEE, pp. 109–118.

Da Silva, L., Borba, P., Mahmood, W., Berger, T., Moisakis, J., 2020. Detecting semantic conflicts via automated behavior change detection. In: 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME). IEEE, pp. 174–184.

Danglot, B., Monperrus, M., Rudametkin, W., Baudry, B., 2020. An approach and benchmark to detect behavioral changes of commits in continuous integration. Empir. Softw. Eng. 25 (4), 2379–2415.

Decan, A., Mens, T., Claes, M., 2017. An empirical comparison of dependency issues in OSS packaging ecosystems. In: 2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER). IEEE, pp. 2–12.

Decan, A., Mens, T., Constantinou, E., 2018a. On the impact of security vulnerabilities in the npm package dependency network. In: International Conference on Mining Software Repositories.

Decan, A., Mens, T., Grosjean, P., 2018b. An empirical comparison of dependency network evolution in seven software packaging ecosystems. Empir. Softw. Eng. URL https://doi.org/10.1007/s10664-017-9589-y.

Dependabot, 2019. Automated dependency updates. https://dependabot.com/ (Accessed on 17 April 2019).

Dietrich, J., Jezek, K., Brada, P., 2014. Broken promises: An empirical study into evolution problems in java programs caused by library upgrades. In: 2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE). IEEE, pp. 64–73.

Dietrich, J., Pearce, D., Stringer, J., Tahir, A., Blincoe, K., 2019. Dependency versioning in the wild. In: 2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR). IEEE, pp. 349–359.

Emanuelsson, P., Nilsson, U., 2008. A comparative study of industrial static analysis tools. Electron. Notes Theor. Comput. Sci. 217, 5–21.

Falleri, J.-R., Morandat, F., Blanc, X., Martinez, M., Monperrus, M., 2014. Fine-grained and accurate source code differencing. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. ACM, pp. 313–324.

Ferrante, J., Ottenstein, K.J., Warren, J.D., 1987. The program dependence graph and its use in optimization. ACM Trans. Programm. Lang. Syst. 9 (3), 319–349.

Fluri, B., Wuersch, M., PInzger, M., Gall, H., 2007. Change distilling: Tree differencing for fine-grained source code change extraction. IEEE Trans. Softw. Eng. 33 (11), 725–743.

Foo, D., Ang, M.Y., Yeo, J., Sharma, A., 2018a. Sgl: A domain-specific language for large-scale analysis of open-source code. In: 2018 IEEE Cybersecurity Development (SecDev). IEEE, pp. 61–68.

Foo, D., Chua, H., Yeo, J., Ang, M.Y., Sharma, A., 2018b. 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. ACM, pp. 791–796.

Fraser, G., Arcuri, A., 2011. Evosuite: automatic test suite generation for object-oriented software. In: Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering, pp. 416–419.

German, D.M., Hassan, A.E., Robles, G., 2009. Change impact graphs: Determining the impact of prior codechanges. Inf. Softw. Technol. 51 (10), 1394–1408.

Gousios, G., Spinellis, D., 2012. Ghtorrent: GitHub's data from a firehose. In: 2012 9th IEEE Working Conference on Mining Software Repositories (MSR). IEEE, pp. 12–21.

`greenkeeper.io`, 2019. Automated dependency management. https://greenkeeper.io/ (Accessed on 17 April 2019).

Gyori, A., Lahiri, S.K., Partush, N., 2017. Refining interprocedural change-impact analysis using equivalence relations. In: Proceedings of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, pp. 318–328.

Hejderup, J., Gousios, G., 2021. Can we trust tests to automate dependency updates? A case study of java projects. Zenodo, URL https://doi.org/10.5281/zenodo.4479015.

Hetzel, W.C., Hetzel, B., 1988. The Complete Guide To Software Testing. QED Information Sciences Wellesley, MA.

Hilton, M., Bell, J., Marinov, D., 2018. A large-scale study of test coverage evolution. In: ASE. pp. 53–63.

IBM Research, 2006. Tj watson libraries for analysis (WALA).

npm Inc., 2018. Details about the event-stream incident. https://blog.npmjs.org/post/180565383195/details-about-the-event-stream-incident (Accessed on 27 November 2018).

Inozemtseva, L., Holmes, R., 2014. Coverage is not strongly correlated with test suite effectiveness. In: Proceedings of the 36th International Conference on Software Engineering. ACM, pp. 435–445.

Just, R., 2014. The major mutation framework: Efficient and scalable mutation analysis for java. In: Proceedings of the 2014 International Symposium on Software Testing and Analysis. pp. 433–436.

Just, R., Jalali, D., Inozemtseva, L., Ernst, M.D., Holmes, R., Fraser, G., 2014. Are mutants a valid substitute for real faults in software testing? In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. ACM, pp. 654–665.

Kikas, R., Gousios, G., Dumas, M., Pfahl, D., 2017. Structure and evolution of package dependency networks. In: Proceedings of the 14th International Conference on Mining Software Repositories. IEEE press, pp. 102–112.

Kochhar, P.S., Lo, D., Lawall, J., Nagappan, N., 2017. Code coverage and postrelease defects: A large-scale study on open source projects. IEEE Trans. Reliab. 66 (4), 1213–1228.

Kropp, N.P., Koopman, P.J., Siewiorek, D.P., 1998. Automated robustness testing of off-the-shelf software components. In: Digest of Papers. Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing (Cat. No. 98CB36224). IEEE, pp. 230–239.

Kula, R.G., German, D.M., Ouni, A., Ishio, T., Inoue, K., 2018. Do developers update their library dependencies? Empir. Softw. Eng. 23 (1), 384–417.

Law, J., Rothermel, G., 2003. Whole program path-based dynamic impact analysis. In: Proceedings of the 25th International Conference on Software Engineering. IEEE Computer Society, pp. 308–318.

Legunsen, O., Shi, A., Marinov, D., 2017. Starts: Static regression test selection. In: 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, pp. 949–954.

Lehnert, S., 2011. A taxonomy for software change impact analysis. In: Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution. ACM, pp. 41–50.

Li, B., Sun, X., Leung, H., 2012. Combining concept lattice with call graph for impact analysis. Adv. Eng. Softw. 53, 1–13.

Li, B., Sun, X., Leung, H., Zhang, S., 2013. A survey of code-based change impact analysis techniques. Softw. Test. Verif. Reliab. 23 (8), 613–646.

Livshits, B., Sridharan, M., Smaragdakis, Y., Lhoták, O., Amaral, J.N., Chang, B.-Y.E., Guyer, S.Z., Khedker, U.P., Mø ller, A., Vardoulakis, D., 2015. In defense of soundiness: a manifesto. Commun. ACM 58 (2), 44–46.

Machalica, M., Samylkin, A., Porth, M., Chandra, S., 2019. Predictive test selection. In: Proceedings of the 41st International Conference on Software Engineering: Software Engineering in Practice. IEEE Press, pp. 91–100.

Mariani, L., Papagiannakis, S., Pezze, M., 2007. Compatibility and regression testing of COTS-component-based software. In: 29th International Conference on Software Engineering (ICSE'07). IEEE, pp. 85–95.

Mezzetti, G., Møller, A., Torp, M.T., 2018. Type regression testing to detect breaking changes in node. js libraries. In: 32nd European Conference on Object-Oriented Programming (ECOOP 2018). Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Mirhosseini, S., Parnin, C., 2017. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In: Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering. IEEE Press, pp. 84–94.

Møller, A., Torp, M.T., 2019. Model-based testing of breaking changes in node. js libraries. Changes 4, 15.

Myers, G.J., Sandler, C., Badgett, T., 2011. The Art of Software Testing. John Wiley & Sons.

Nguyen, H.A., Nguyen, T.N., Dig, D., Nguyen, S., Tran, H., Hilton, M., 2019. Graph-based mining of in-the-wild, fine-grained, semantic code change patterns. In: Proceedings of the 41st International Conference on Software Engineering. IEEE Press, pp. 819–830.

npm, 2018. How to use semantic versioning. https://docs.npmjs.com/getting-started/semantic-versioning (Accessed on 21 October 2018).

Orso, A., Apiwattanapong, T., Harrold, M.J., 2003. Leveraging field data for impact analysis and regression testing. In: ACM SIGSOFT Software Engineering Notes, vol. 28. (5), ACM, pp. 128–137.

Orso, A., Apiwattanapong, T., Law, J., Rothermel, G., Harrold, M.J., 2004. An empirical comparison of dynamic impact analysis algorithms. In: Proceedings of the 26th International Conference on Software Engineering. IEEE Computer Society, pp. 491–500.

Papadakis, M., Kintis, M., Zhang, J., Jia, Y., Le Traon, Y., Harman, M., 2019. Mutation testing advances: an analysis and survey. In: Advances in Computers, vol. 112. Elsevier, pp. 275–378.

Ponomarenko, A., 2011. Java API compliance checker. https://lvc.github.io/japi-compliance-checker/(Accessed on 17/04/2019).

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

Raemaekers, S., van Deursen, A., Visser, J., 2017. Semantic versioning and impact of breaking changes in the maven repository. J. Syst. Softw. 129, 140–158.

Reif, M., Kübler, F., Eichberg, M., Helm, D., Mezini, M., 2019. Judge: identifying, understanding, and evaluating sources of unsoundness in call graphs. In: Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. ACM, pp. 251–261.

Ren, X., Shah, F., Tip, F., Ryder, B.G., Chesley, O., 2004. Chianti: a tool for change impact analysis of java programs. In: ACM Sigplan Notices, vol. 39. (10), ACM, pp. 432–448.

Renovate, 2019. Automated dependency management. https://renovatebot.com/ (Accessed on 26/07/2019).

Ryder, B.G., Tip, F., 2001. Change impact analysis for object-oriented programs. In: Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering. ACM, pp. 46–53.
Shi, A., Hadzi-Tanovic, M., Zhang, L., Marinov, D., Legunsen, O., 2019. Reflection-aware static regression test selection. Proceedings of the ACM on Programming Languages 3 (OOPSLA), 1–29.
Strobel, M., 2016. Procyon/java decompiler.
Tip, F., 1994. A Survey of Program Slicing Techniques. Centrum voor Wiskunde en Informatica.
Trockman, A., Zhou, S., Kästner, C., Vasilescu, B., 2018. Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem. In: Proceedings of the 40th International Conference on Software Engineering. ACM, pp. 511–522.
Whittaker, J.A., 2002. How To Break Software: A Practical Guide To Testing with Cdrom. Addison-Wesley Longman Publishing Co., Inc..
Wittern, E., Suter, P., Rajagopalan, S., 2016. A look at the dynamics of the JavaScript package ecosystem. In: Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on. IEEE, pp. 351–361.
Yoo, S., Harman, M., 2012. Regression testing minimization, selection and prioritization: a survey. Softw. Test. Verif. Reliab. 22 (2), 67–120.
Zeller, A., 2009. Why Programs Fail: A Guide To Systematic Debugging. Elsevier.
Zhang, J., Zhang, L., Harman, M., Hao, D., Jia, Y., Zhang, L., 2018. Predictive mutation testing. IEEE Trans. Softw. Eng..

Joseph Hejderup is a Ph.D. student at the Delft University of Technology, the Netherlands. His primary research interest is to make package management systems more intelligent, safe, and robust using program analysis and empirical methods. He is the main author of Präzi, a technique that constructs fine-grained dependency networks using call graphs. He holds an M.Sc. from the Delft University of Technology, the Netherlands.

Georgios Gousios, is a research engineer at Facebook and an associate professor at the Delft University of Technology, The Netherlands (on leave). He works in the fields of software analytics, software ecosystems, software processes, and machine learning for software engineering. He is the main author of the GHTorrent data collection and curation framework and various widely used tools and datasets.