问题
- suggesting upgrades
- 如何检查开源项目的API不兼容的问题

Efficient Static Checking of Library Updates

Darius Foo
CA Technologies
Singapore
darius.foo@ca.com

Hendy Chua
CA Technologies
Singapore
hendy.chua@ca.com

Jason Yeo
CA Technologies
Singapore
jason.yeo@ca.com

Ming Yi Ang
CA Technologies
Singapore
mingyi.ang@ca.com

Asankhaya Sharma
CA Technologies
Singapore
asankhaya.sharma@ca.com

ABSTRACT

Software engineering practices have evolved to the point where a developer writing a new application today doesn't start from scratch, but reuses a number of open source libraries and components. These third-party libraries evolve independently of the applications in which they are used, and may not maintain stable interfaces as bugs and vulnerabilities in them are fixed. This in turn causes API incompatibilities in downstream applications which must be manually resolved. Oversight here may manifest in many ways, from test failures to crashes at runtime. To address this problem, we present a static analysis for automatically and efficiently checking if a library upgrade introduces an API incompatibility.

Our analysis does not rely on reported version information from library developers, and instead computes the actual differences between methods in libraries across different versions. The analysis is scalable, enabling real-time diff queries involving arbitrary pairs of library versions. It supports a vulnerability remediation product which suggests library upgrades automatically and is lightweight enough to be part of a continuous integration/delivery (CI/CD) pipeline. To evaluate the effectiveness of our approach, we determine semantic versioning adherence of a corpus of open source libraries taken from Maven Central, PyPI, and RubyGems. We find that on average, 26% of library versions are in violation of semantic versioning. We also analyze a collection of popular open source projects from GitHub to determine if we can automatically update libraries in them without causing API incompatibilities. Our results indicate that we can suggest upgrades automatically for 10% of the libraries.

CCS CONCEPTS

• **Software and its engineering** → **Software evolution**; **Maintaining software**; *Automated static analysis*;

KEYWORDS

automated remediation, library upgrades, call graphs, api diffs, semantic versioning

ACM Reference Format:
Darius Foo, Hendy Chua, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. 2018. Efficient Static Checking of Library Updates. In *Proceedings of the 26th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE '18), November 4–9, 2018, Lake Buena Vista, FL, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3236024.3275535

1 INTRODUCTION

The use of open-source and third-party components has increased in the development of software. Centralized package distribution systems like Maven Central for Java, RubyGems for Ruby, and PyPI for Python make the task of downloading and using these components very convenient for the average developer. However, because third-party libraries evolve independently of the applications that use them, bugs and vulnerabilities in third-party components are hard to trace and fix in downstream applications. Even when vulnerabilities can be fixed by updating to a newer version of a library, there can be API incompatibilities with downstream applications which must be manually resolved. Oversight here may manifest in many ways, from test failures to crashes at runtime.

One solution is semantic versioning[1] (SemVer), a structured versioning scheme which relies on adherence to conventions. Library authors number versions with triplets of the form MAJOR.MINOR.PATCH, and release a major version when introducing a backward-incompatible API change to indicate this to downstream applications. As this versioning scheme is structured, version numbers can be operated on by tools – an example is Bundler's ~> operator, which only upgrades packages across patch version boundaries. However, the compliance of *source code* to the scheme must be manually enforced, and it has been criticized as inadequately able to capture the nuances of change in software development [1].

To offer a better solution, we present a static analysis for automatically and efficiently checking if a library upgrade introduces an API compatibility (colloquially termed a *breaking change*). We do not rely on a surrogate source of truth such as semantic versioning, and instead statically compute the differences between source-level elements of the library (in particular, methods and functions), also

[1] https://semver.org/

ESEC/FSE '18, November 4–9, 2018, Lake Buena Vista, FL, USA

ACM ISBN 978-1-4503-5573-5/18/11...$15.00
https://doi.org/10.1145/3236024.3275535

taking control flow into account. Our analysis is scalable, enabling real-time diff queries involving arbitrary pairs of library versions.

Our solution forms the basis for a new feature in the CA Veracode Software Composition Analysis product which suggests library upgrades automatically and is lightweight enough to be part of a continuous integration/delivery (CI/CD) pipeline [17]. Left to their own devices, developers do not update dependencies [11], as it is seen as "extra effort and added responsibility", and the downsides of failing to do so are not as visible. We believe that the ability to have upgrades automatically carried out – but not carelessly, *with guarantees about their effects* – would go a long way towards eliminating easily-preventable classes of mistakes and vulnerabilities. A recent study on automated pull requests [14] reached the same conclusions, finding that automation caused a 60% increase in upgrades, and that notification fatigue and concerns about breaking changes became the bottleneck thereafter. Our analysis directly address the problem of breaking changes in that we can statically show the user which library upgrades are possible without introducing API incompatibilities in their application.

To evaluate the effectiveness of our approach, we determine semantic versioning adherence of a corpus of open source libraries collected from our customer scan data. These libraries cover three different languages from their respective central repositories – Java (Maven Central), Python (PyPI), and Ruby (RubyGems). We find that on average, 26% of library versions are in violation of semantic versioning, i.e. they break backward compatibility without a major version update. We also analyze a sample of popular open source GitHub projects to determine the prevalence of API incompatibilities in practice. We find that, using our static analysis, we can automatically suggest upgrades for 10% of the libraries in these open source projects.

Our main technical contributions are:

- A static analysis that detects breaking changes in libraries accurately, allowing upgrades to be suggested.
- A novel method of composing diffs which enables diff queries on arbitrary library version pairs to be answered in real time, at the cost of linear space (instead of quadratic).
- A case study of open source libraries published on Maven Central, PyPI, and RubyGems to assess adherence to semantic versioning. On average, 72% of libraries violate SemVer in some version, and 26% of all library versions violate SemVer.

2 RELATED WORK

2.1 Automated Library Upgrades

Prior work suggesting automated upgrades ranges from following simple rules, such as always updating all dependencies within their constraints and relying on test suites to check for breakage[2,3], to sophisticated attempts to actually patch APIs or dependent code [4, 8, 21] by inferring new API usage from examples. In contrast, our approach statically computes diffs to check for breakage and automatically update libraries that do not cause incompatiblities.

Other static analysis approaches for analyzing library upgrades make use of dependency analysis [19], symbolic execution [3] and JML contracts [23] to model the semantics of changes.

[2]https://www.deppbot.com/
[3]https://greenkeeper.io/

2.2 Structured Diffs

Textual, subsequence-based diffs, such as those produced by the Unix *diff* tool, are widely used for visually comparing program fragments and sharing patches. They can be computed quickly, but do not take into account programming language syntax, making them unsuitable for applications such as automated program repair.

Syntactic diffs utilize syntax to ignore textual details. They may be computed at multiple levels of abstraction: syntactic API diffs include only program elements intended for external use, such as classes and methods. Examples are the documents published by Apple[4] and Google[5] to summarize differences between versions of their mobile APIs. UMLDiff [20], GumTree [6], and Wuu Yang [22] are implementations of syntactic diffs which compare the syntax trees of source code instead of only interfaces.

Diffs may also reflect *semantic* information, such as control flow and state [9]. Semantic Diff [10] computes differences in the observable input-output behaviour of procedures by reasoning about dependencies between variables. SymDiff [12] checks for *partial equivalence* between programs – terminating executions with identical inputs and outputs. Mezzetti et al. [13] use a dynamic analysis based on the test suites of dependent libraries to infer library interfaces for comparison. Our approach sits in this category, as we compute syntactic API diffs that are enriched with control flow information, applying diff composition thereafter to compare arbitrary versions of a given library.

2.3 SemVer Compliance

We conduct a case study of three open source ecosystems: Maven Central, RubyGems, and PyPI to evaluate the adherence of library developers to SemVer scheme. Prior work in this area by Raemaekers et al. [16] goes into much greater detail, but only for Maven Central. Other related studies evaluate breaking changes in open source projects on npm [13] and CRAN [2].

3 APPROACH

3.1 Basic Diffs

Consider the problem of computing diffs for library APIs. We begin by computing a minimal, language-agnostic representation of a library's API, which we term a *signature*. The representation we use is a set of tuples of an *identifier* and a *hash*.

The identifiers give canonical names to the program elements in libraries that we wish to compare. For example, the methods of an object-oriented API, are represented as a tuple of module, class, method names and an argument descriptor.

The hashes summarize the content of the program element, allowing us to quickly determine if it has changed – for example, we hash the bytecode of Java methods, eliding syntactic features such as variable names, but including literals and instructions that affect control flow.

Given the signatures of two libraries, we use Myers' algorithm [15] to compute a *diff*: an edit script relating them, with the slight modification that we key elements by identifier instead of position

[4]https://developer.apple.com/library/archive/releasenotes/General/iOS10APIDiffs/index.html
[5]https://developer.android.com/sdk/api$_d$iff/p-dp1/changes

```
class A {
  public int a() {
    return 2;
  }
  public int b(int x) {
    return x + 3;
  }
}
```

Figure 1: Example: v_1

```
class A {
  // Method a deleted
  public int b(int x) {
    return x + 2; // Modified
  }
  public int c() { // Inserted
    return 1;
  }
}
```

Figure 2: Example: v_2

Table 1: Example: computed diff from v_1 to v_2

method	operation
A.a()I	**DELETE**
A.b(I)I	**CHANGE**
A.c()I	**INSERT**

in a sequence. This gives us a set of tuples of identifier and *diff operation*, where the latter is one of the symbols **INSERT**, **DELETE**, or **CHANGE**. **CHANGE**s are opaque as program content is summarized using a hash. In particular, we do not perform a more granular syntax-tree-based diff, like GumTree [6] does.

Since diffs are meant to relate the *public APIs* of libraries, as a post-processing step we exclude program elements which we know not to be publicly accessible. In Java, this information is explicit; for dynamically-typed languages such as Python and Ruby, we rely on heuristics (based on variable names and statically-known calls to methods that modify access) to exclude such elements.

As a running example, consider two versions of a Java class (Figures 1 and 2). The computed set of diff operations between these two versions is shown in Table 1.

3.2 Transitively-changed Methods

Hashing methods gives us an approximate way to detect changes, but processing methods in isolation (i.e. ignoring inter-procedural control flow) causes transitive changes to methods to be missed. For example, if public method m_1 calls private method m_2 and only m_2 has changed, we would miss the fact that m_1's semantics is now different if we exclude m_2 due to its private access. This is illustrated in Figure 3.

To solve this problem, we first build call graphs of library programs – our call graph construction algorithm uses standard techniques of CHA [5] and VTA [18] and is deployed as part of the CA Veracode Software Composition Analysis product. We then use the call graphs to improve diffs: given a method m whose identifier is present in a diff, we also include all public methods that call m. m may or may not be later dropped from the diff, depending on if it is public, but we will no longer exclude its public callers.

```
class A { // Before
  public int m1(int x) {
    return m2(x);
  }
  private int m2(int y) {
    return y + 1;
  }
}

class A { // After
  public int m1(int x) {
    // Syntactically unchanged, but
    // returns a different result and
    // so should have a different hash
    return m2(x);
  }
  private int m2(int y) {
    return y + 2; // Changed
  }
}
```

Figure 3: Example: transitive changes

Table 2: Example: computed diff from v_2 to v_3

method	operation
A.a()I	**INSERT**
A.b(I)I	**CHANGE**
A.c()I	**DELETE**

3.3 Fast Queries

Call graph construction is thus necessary for the accuracy of diffs, but imposes nontrivial overhead. For instance, the largest libraries on Maven Central may take hours to completely analyze. This makes it infeasible to compute accurate diffs on demand, say as part of an automated library upgrade step in a CI/CD pipeline. In this section, we describe a means of precomputing information that enables real-time diff queries on arbitrary pairs of library versions.

A naive approach would be to precompute and store every pair of diffs, consuming space that grows quadratically with the number of versions. This is unlikely to scale since real-world libraries have hundreds of versions, and in general libraries may have up to one version *per commit*[6]. Doing this for a commonly-accessed subset of libraries and a window of recent versions is feasible, but shifts the problem to determining this subset, and of course only works for versions within the subset.

Our approach is to precompute diffs between only *consecutive pairs* of library versions, then compose individual diffs to derive diffs for arbitrary version ranges. This strikes a good balance, requiring a linear amount of space and running quickly enough in practice.

3.4 Diff Composition

We begin with a concrete example of diff composition. Consider the diffs given in Table 1 and Table 2, assuming they are computed

[6] https://mvnrepository.com/artifact/com.lihaoyi/ammonite-terminal

Table 3: Example: composed diff from v_1 to v_3

method	operation
A.a()I	**CHANGE**
A.b(I)I	**CHANGE**
A.c()I	**MISSING**

```
data State = Absent | Present

data Diff : State -> State -> Type where
    Insert : Diff Absent Present
    Change : Diff Present Present
    Delete : Diff Present Absent
    Unchanged : Diff Present Present
    Missing : Diff Absent Absent
```

Figure 4: Modeling diffs as a type

from consecutive library versions v_1 and v_2, and v_2 and v_3, respectively. Method *a* was **DELETE**d, then re**INSERT**ed: because we no longer have access to its content, we must assume conservatively that the reinsertion is different, and that it has **CHANGE**d overall. Method *b* was **CHANGE**d in both diffs: we can say no more than that it is **CHANGE**d as well. Finally, method *c* was **INSERT**ed, then **DELETE**d: it is now **MISSING** (an operation we have not yet defined), and we must treat it as such in any further composition of the resulting diff. The final, composed diff is given in Table 3. We see also that it is impossible for a diff between v_3 to v_4 to have that method *c* is **DELETE**d (given that it was never present in v_3 to begin with), suggesting that composition is partial.

We now formalize diff composition. As defined earlier, a diff is a set of tuples of a method identifier and some *diff operation* which describes how the method changed across versions.

There are *5* primitive diff operations – we saw **INSERT**, **DELETE**, and **CHANGE** earlier, and introduce two more for explicitness: **UNCHANGED**, which means that a method appears unchanged across versions, and **MISSING**, for when it is missing altogether. The latter two operatons are never produced when diffs are computed, but we include (and distinguish) them because certain compositions of operations are absurd, e.g. **INSERT** followed by **INSERT**. The validity of an operation relating two methods across versions depends on the methods' states: e.g. a method must be *absent* for an **INSERT**ion of it to make sense, and it must be *present* after. **UNCHANGED** and **MISSING** relate methods with different states.

We may represent the diff operations collectively as a type indexed by the states of the method they relate *before* and *after* they are applied. Each diff operation becomes an inhabitant of this type. This is shown in Figure 4.

The diff composition function must then have the following type:

```
compose : Diff a b -> Diff b c -> Diff a c
```

In this way, we partially reduce the problem of checking the validity of a particular composition to determining how the operands of `compose` constrain its result.

`compose` is uniquely defined on many inputs. For example, `compose Delete Missing` *must* be `Delete`; no other operation is well-typed. The fact that compositon is partial is reflected in the fact that illegal compositions such as `compose Insert Insert` are not well-typed.

Table 4: Diff composition function

	I	**C**	**D**	**U**	**M**
I	⊥	I	M	I	⊥
C	⊥	C	D	C	⊥
D	C	⊥	⊥	⊥	D
U	⊥	C	D	U	⊥
M	I	⊥	⊥	⊥	M

Table 5: Conflated diff composition function

	I	**C**	**D**	**UM**
I	⊥	I	UM	I
C	⊥	C	D	C
D	C	⊥	⊥	D
UM	I	C	D	UM

The only ambiguity arises when selecting between **CHANGE** and **UNCHANGED**; as we do not model hashes in our types, they have the same type `Diff Present Present` (only methods which are present throughout may be said to have changed or remained unchanged). We resolve the ambiguity manually, choosing **CHANGE** where possible as it is more conservative than **UNCHANGED**.

The final composition function is given in Table 4. Rows are the first argument and columns are the second. We represent ill-typed combinations with the ⊥ symbol. Composition is not symmetric:

```
compose Insert Delete = Missing
compose Delete Insert = Change
```

However, it is associative (which can proven by exhaustion).

3.5 Conflating Operations

It turns out that we can conflate **UNCHANGED** and **MISSING** into a single operation, **UNKNOWN** (abbreviated **UM**), since they occur in mutually exclusive scenarios. This is useful in practice. Say we diff a list of 100 items against a list of 101; we would want to store a single **INSERT** instead of also storing 100 **UNCHANGED**s. Defaulting to **UNKNOWN** when an item is absent allows us to store the information concisely. This doesn't change composition semantics (proven by exhaustion).

Implementing this change gives us the new function, in Table 5. The time complexity of diff composition is linear in the number of library methods and versions, like diff computation itself.

3.6 Suggesting Upgrades

We then use diffs to suggest upgrades and determine if they induce breaking changes. In the CA Veracode Software Composition Analysis product, we identify vulnerable versions of a library and suggest upgrades to fix those versions. Given a library at version v_1 and the set of versions v_s of the same library, we choose another version from v_s which succeeds v_1 and does not possess vulnerabilities[7] associated with v_1. This may be done using various heuristics and may be further optimized; we currently choose the closest version to minimize diff size and induce the fewest breaking changes.

Given the library diff and the pair of library versions involved in the upgrade v_{from} and v_{to}, we restrict it to only **DELETE** and **CHANGE** operations; this is exactly the set of methods in v_{from}

[7] Vulnerability data is assumed to come from an external source, such as NVD or a proprietary vulnerability database [24]

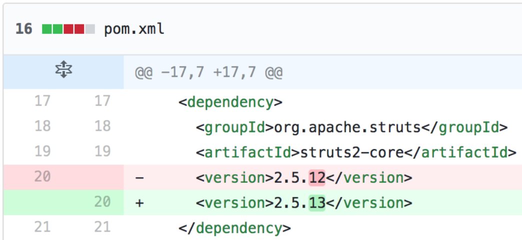

Figure 5: Generated patch to pom.xml

Table 6: Report in generated pull request

Type	Library	From	To	Breaking
MAVEN	org.apache.struts struts2-core	2.5.12	2.5.13	No
MAVEN	net.bull.javamelody javamelody-core	1.59.0	1.62.0	Possibly

whose callers will be affected by the upgrade. We then check the call graphs we compute from user code to see if they are calling any affected methods; if so, we consider the upgrade to be a breaking change. We use SGL, a domain-specific language for program analysis, to perform the call graph traversal. The specifics of SGL are described in [7] and are out of scope for this work.

The next step is to generate a patch, rendered in Figure 5 as a GitHub pull request. Since we patch files typically maintained by hand, we take care to minimize changes by parsing the files, then using position information to make granular edits. An example of what we include in generated pull requests is shown in Table 6: we specify the to- and from-versions of the upgrade and say whether or not we could statically determine that the change was breaking.

4 EXPERIMENTS AND EVALUATION

4.1 SemVer Compliance

We analyzed 5106 libraries (based on our customer scan data) across three popular open source library repositories: Maven Central (3273 libraries), RubyGems (1332), and PyPI (501), computing diffs for 114,199 library versions in total.

Our results indicate that on average, 72% of libraries violate SemVer in *some* version: the actual numbers are 80% (RubyGems), 67% (Maven Central), and 82% (PyPI). In addition, on average, 26% of all library versions are in violation, actual numbers being 31% (RubyGems), 24% (Maven Central), and 31% (PyPI). The figure for Maven Central agrees with prior work [16], which puts the number of violating versions between 28.4% and 23.7% over time. The overall distribution of violations is shown in Figure 6, as a plot of the number of libraries (y-axis) that have a given percentage of versions within them in violation (x-axis).

As a concrete example, consider the popular `requests` library, between versions 2.3.0 and 2.4.0 – a minor version bump, over which breaking changes are not expected – `requests.structures.IteratorProxy` was deleted[8]. For this not to be considered a SemVer violation, `IteratorProxy` must not be part of `requests`' pubilc API, however it is difficult to determine this using a static analysis (as nothing prevents one from importing `requests.structures`).

[8]https://github.com/requests/requests/compare/v2.3.0...v2.4.0#diff-2bdbe7e19f5215e8c319573cdd114f01L16

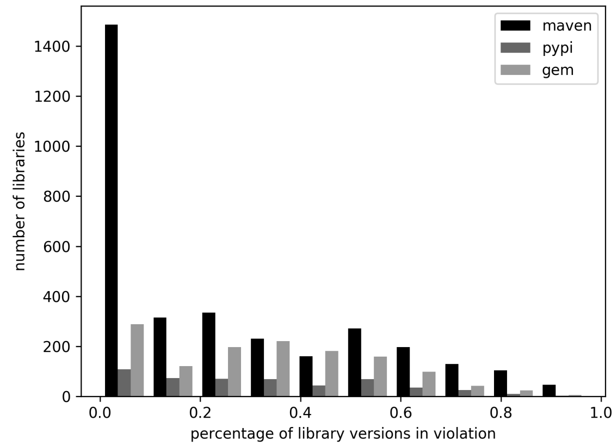

Figure 6: Semantic versioning compliance

Table 7: API incompatibilities on GitHub

	Java	**Python**	**Ruby**
Projects	274	422	503
Direct dependencies	4777	2572	4096
Direct vulnerable dependencies	246	110	250
Suggested upgrades	150	64	123
Non-breaking	28 (19%)	0 (0%)	7 (6%)

Consulting human-readable sources like commit messages and the change log also yielded nothing in this case, so we assume it to be a breaking change. The fact that such mistakes may slip through is why a manually-enforced convention like SemVer is insufficient to ensure reliable upgrades.

4.2 API Incompatibilities in Open Source Projects

We analyzed a collection of popular open source projects from GitHub to determine the impact of API incompatibilities on suggesting upgrades automatically. A dynamic dependency analysis was first performed to identify libraries included: e.g. as shown in Table 7, in Java we identified 4777 unique direct dependencies, of which 246 had known vulnerabilities in our database. We were then able to suggest upgrades for 150 of those; 36 had no safe versions to upgrade to, and 60 failed due to the errors in an earlier part of the pipeline (such as malformed class files or the inability to successfully compute a call graph). Of the 150 upgrades, we are able to statically show that 28 (18.7%) are non-breaking.

Across languages, 10% of upgrades are non-breaking on average. We believe the notably-lower numbers for Python and Ruby are due to the difficulties of static call graph construction for those languages and the false positives that result from over-approximation.

5 DISCUSSION

5.1 Threats to Validity

To handle large real-world libraries as part of a CI/CD product, our static analysis techniques rely on a number of approximations which may introduce unsoundness. They are listed in this section.

Limitations of Static Analysis. A call graph computed using CHA/VTA over-approximates the dynamic control flow of a program, causing false positives: call graph edges that are never traversed at runtime.

Separately, bytecode hashing is used to determine if methods have changed. This results in false positives if methods differ syntactically, but are semantically unchanged, e.g. due to refactoring changes, or because different (versions of) compilers may emit different instruction sequences for the same piece of source code.

Unsupported Language Features. False negatives also occur due to lack of support for dynamic language features such as reflection and classpath introspection. These manifest as missing call graph edges, causing potentially-breaking changes to be missed.

Computing Diffs in Isolation. Call graphs and diffs are computed for single libraries at a time. If a version of a library v_a depends on v_b, and the version that comes after v_a (denoted v'_a) depends on v'_b, we will fail to pick up potential breaking changes due to *calls* to methods which now have different semantics *due to the transitive upgrade from* v_b *to* v'_b.

Insufficient Semantic Information. There is also the issue of insufficient semantic information being statically present in source code. An example is the access levels of methods in Python, which are mostly implicit – while the runtime does treat underscore-prefixed names specially in some contexts, it is mostly a convention. Whether or not an API is meant to be internal is not always deducible from source code, and so we necessarily over-approximate, employ heuristics, and guess when analyzing such programs.

6 CONCLUSION AND FUTURE WORK

We presented a static analysis for computing diffs between libraries and determining if an upgrade introduces an API incompatibility. It is scalable, supporting real-time diff queries involving arbitrary pairs of library versions. This makes it lightweight enough to be part of a continuous integration/delivery (CI/CD) pipeline, enabling a vulnerability remediation product that is able to automatically upgrade libraries in 10% of cases. We also evaluated adherence to the semantic versioning scheme on Maven Central, PyPI, and RubyGems, finding that 26% of library versions are in violation.

For future work, we aim to improve accuracy and lower false positive rates. A dynamic analysis could be combined with static call graphs to improve their accuracy, and usability and false negative rates could both be improved by accounting for more language features. Another direction is to make results more actionable by generating test cases which exercise code paths which we know to be involved in a breaking upgrade, or optimizing the selection of upgrade versions to minimize the manual work users must perform.

REFERENCES

[1] 2014. Why Semantic Versioning Isn't. https://gist.github.com/jashkenas/cbd2b088e20279ae2c8e
[2] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 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 (FSE 2016)*. ACM, New York, NY, USA, 109–120. https://doi.org/10.1145/2950290.2950325
[3] Marcel Böhme, Abhik Roychoudhury, and Bruno C. d. S. Oliveira. 2013. Regression Testing of Evolving Programs. *Advances in Computers* 89 (2013), 53–88.
[4] Barthelemy Dagenais and Martin P. Robillard. 2009. SemDiff: Analysis and Recommendation Support for API Evolution. In *Proceedings of the 31st International Conference on Software Engineering (ICSE '09)*. IEEE Computer Society, Washington, DC, USA, 599–602. https://doi.org/10.1109/ICSE.2009.5070565
[5] Jeffrey Dean, David Grove, and Craig Chambers. 1995. Optimization of object-oriented programs using static class hierarchy analysis. In *European Conference on Object-Oriented Programming*. Springer, 77–101.
[6] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. ACM, 313–324.
[7] Darius Foo, Jason Yeo, Ming Yi Ang, and Asankhaya Sharma. 2018. SGL: A domain-specific language for large-scale analysis of open-source code. *IEEE Cybersecurity Development, SecDev* (2018).
[8] Johannes Henkel and Amer Diwan. 2005. CatchUp!: Capturing and Replaying Refactorings to Support API Evolution. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*. ACM, New York, NY, USA, 274–283. https://doi.org/10.1145/1062455.1062512
[9] Susan Horwitz. 1990. Identifying the Semantic and Textual Differences Between Two Versions of a Program. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation (PLDI '90)*. ACM, New York, NY, USA, 234–245. https://doi.org/10.1145/93542.93574
[10] Daniel Jackson, David A Ladd, et al. 1994. Semantic Diff: A Tool for Summarizing the Effects of Modifications.. In *ICSM*, Vol. 94. 243–252.
[11] Raula Gaikovina Kula, Daniel M German, Ali Ouni, Takashi Ishio, and Katsuro Inoue. [n. d.]. Do developers update their library dependencies? *Empirical Software Engineering* ([n. d.]), 1–34.
[12] Shuvendu K Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *International Conference on Computer Aided Verification*. Springer, 712–717.
[13] Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. [n. d.]. Type Regression Testing to Detect Breaking Changes in Node.js Libraries. ([n. d.]).
[14] Samim Mirhosseini and Chris Parnin. 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, 84–94.
[15] Eugene W Myers. 1986. An O(ND) difference algorithm and its variations. *Algorithmica* 1, 1-4 (1986), 251–266.
[16] S. Raemaekers, A. van Deursen, and J. Visser. 2014. Semantic Versioning versus Breaking Changes: A Study of the Maven Repository. In *2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*. 215–224. https://doi.org/10.1109/SCAM.2014.30
[17] Mojtaba Shahin, Muhammad Ali Babar, and Liming Zhu. 2017. Continuous integration, delivery and deployment: a systematic review on approaches, tools, challenges and practices. *IEEE Access* 5 (2017), 3909–3943.
[18] Vijay Sundaresan, Laurie Hendren, Chrislain Razafimahefa, Raja Vallée-Rai, Patrick Lam, Etienne Gagnon, and Charles Godin. 2000. *Practical virtual method call resolution for Java*. Vol. 35. ACM.
[19] Mohsen Vakilian, Raluca Sauciuc, J David Morgenthaler, and Vahab Mirrokni. 2015. Automated decomposition of build targets. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, Vol. 1. IEEE, 123–133.
[20] Zhenchang Xing and Eleni Stroulia. 2005. UMLDiff: An Algorithm for Object-oriented Design Differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE '05)*. ACM, New York, NY, USA, 54–65. https://doi.org/10.1145/1101908.1101919
[21] Z. Xing and E. Stroulia. 2007. API-Evolution Support with Diff-CatchUp. *IEEE Transactions on Software Engineering* 33, 12 (Dec 2007), 818–836. https://doi.org/10.1109/TSE.2007.70747
[22] Wuu Yang. 1991. Identifying syntactic differences between two programs. *Software: Practice and Experience* 21, 7 (1991), 739–755.
[23] Jooyong Yi, Dawei Qi, Shin Hwei Tan, and Abhik Roychoudhury. 2013. Expressing and checking intended changes via software change contracts. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*. ACM, 1–11.
[24] Yaqin Zhou and Asankhaya Sharma. 2017. Automated identification of security issues from commit messages and bug reports. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 914–919.