Efficient Static Checking of Library Updates

Darius Foo, Hendy Chua, Jason Yeo, Ang Ming Yi, Asankhaya Sharma
Motivation

- Applications depend on lots of libraries

For each Java library depended on, 4 others are added

For each JS library depended on, 9 others are added
Motivation

- Libraries evolve, and we’d like to keep up
  - Security patches
  - Bug fixes
  - New features
Motivation

- Upgrades are hard!
  - Compile errors
  - API incompatibilities
  - Test failures
  - Dependency conflicts
  - Crashes at runtime
  - Subtle changes in behavior
Motivation

- Semantic versioning (SemVer)
  - Adherence to conventions; MAJOR.MINOR.PATCH
    - Backwards-incompatible change: bump MAJOR
    - Backwards-compatible addition: bump MINOR
    - Backwards-compatible bug fix: bump PATCH
  - Structured; tooling-friendly
    - ~> operator
  - Inadequately able to capture nuances of change
  - Compliance of source code to scheme must be manually enforced
What we want

- Automated, safe library upgrades
- Automated pull requests, but with guarantees
    - 60% increase in frequency of upgrades
    - Notification fatigue and concerns about breaking changes became bottleneck thereafter
- Fast enough to run in a CI pipeline
Approach

- Static analysis for detecting API incompatibilities in upgrades
- Compute differences between source-level elements
  - Methods, functions
- Take control flow into account
- Determine if code to be upgraded is calling a changed/deleted method
- Precompute diffs and compose them on request
Related work

- Automated library upgrades
  - Deppbot, Greenkeeper
    - Update all dependencies within constraints and rely on test suites
  - SemDiff (Dagenais, et al.), Diff-CatchUp (Xing, et al.)
    - Recommend replacements for changed methods by looking at how libraries adapt to their own changes
  - CatchUp! (Henkel, et al.)
    - Capture refactoring actions on an API, replay them on uses of an API
Related work

- Structured diffs
  - Textual, subsequence-based diffs (diff)
    - Computed quickly, but without considering syntax
  - Syntactic diffs
    - Computed between syntactic elements
    - Google’s Android API diffs
    - UMLDiff, Gumtree
  - Semantic diffs
    - Reflects control flow, state
    - Semantic Diff: computes differences in input-output behaviour of functions
    - SymDiff: partial equivalence between programs
Related work

- SemVer compliance
  - Similar studies for npm and CRAN
Approach

- Basic API diffs
  - Extract tuples of method name and bytecode hash
  - Hashes approximate method implementations and detect changes
  - Libraries are sets of methods

<table>
<thead>
<tr>
<th>Method name</th>
<th>Hash</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.example.A.a([B)l</td>
<td>0xCAFEBEEF</td>
</tr>
</tbody>
</table>
Approach

- Basic API diffs
  - Given two library signatures, use Myers’ algorithm to compute a diff
  - Three operations: INSERT, CHANGE, DELETE
  - Drop non-public methods to get the API diff

<table>
<thead>
<tr>
<th>Method name</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>com.example.A.a([B)l</td>
<td>DELETE</td>
</tr>
</tbody>
</table>
Approach

- Basic API diffs

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

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

<table>
<thead>
<tr>
<th>Method</th>
<th>Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.a()</td>
<td>DELETE</td>
</tr>
<tr>
<td>A.b()</td>
<td>CHANGE</td>
</tr>
<tr>
<td>A.c()</td>
<td>INSERT</td>
</tr>
</tbody>
</table>
Approach

- **Transitively-changed methods**
  - Dropping private methods is problematic...
    - (Public) $m_1$ is unchanged, but calls (private) $m_2$, which has changed
    - Changes to $m_1$ are lost after dropping $m_2$

```java
class A {
    public int m1(int x) {
        return m2(x);
    }

    private int m2(int y) {
        return y + 1;
    }
}
```
Approach

- Transitively-changed methods
  - Build call graphs and use them to improve diffs
  - Transitive callers of changed/deleted methods must also have changed
  - Private methods may still be dropped, but we no longer lose changes
Approach

- Fast queries
  - How to compute diffs on demand?
  - Call graph construction is expensive
    - Hours for largest libraries on Maven Central
  - Precompute and store every pair of diffs?
    - $O(v^2)$ space
    - Real-world libraries have *hundreds* of versions
  - Do this for a subset of libraries?
    - How to determine this subset?
    - Does not work outside it
Approach

- Diff composition
  - Precompute diffs only between consecutive pairs of library versions
  - Compose diffs to derive diffs for arbitrary version ranges
  - Linear space, linear time

\[ v_1 \quad d_1 \quad v_2 \quad d_2 \quad v_3 \quad d_1 \circ d_2 \]
Approach

- Diff composition
  - DELETE • INSERT = ?
Approach

- Diff composition
  - DELETE ◦ INSERT = ?
    - CHANGE: assume conservatively that reinsertion is different
Approach

- Diff composition
  - DELETE ◦ INSERT = ?
    - CHANGE: assume conservatively that reinsertion is different
  - CHANGE ◦ CHANGE = ?
Approach

- Diff composition
  - DELETE ◦ INSERT = ?
    - CHANGE: assume conservatively that reinsertion is different
  - CHANGE ◦ CHANGE = ?
    - CHANGE: can’t say any more
Approach

- **Diff composition**
  - DELETE ◦ INSERT = ?
    - CHANGE: assume conservatively that reinsertion is different
  - CHANGE ◦ CHANGE = ?
    - CHANGE: can’t say any more
  - INSERT ◦ DELETE = ?
Approach

- Diff composition
  - DELETE ◦ INSERT = ?
    - CHANGE: assume conservatively that reinsertion is different
  - CHANGE ◦ CHANGE = ?
    - CHANGE: can’t say any more
  - INSERT ◦ DELETE = ?
    - MISSING (not yet defined)
Approach

- **Diff composition**
  - DELETE ◦ INSERT = ?
    - CHANGE: assume conservatively that reinsertion is different
  - CHANGE ◦ CHANGE = ?
    - CHANGE: can’t say any more
  - INSERT ◦ DELETE = ?
    - MISSING (not yet defined)
  - DELETE ◦ DELETE = ?
  - INSERT ◦ INSERT = ?
Approach

- Diff composition
  - DELETE ◦ INSERT = ?
    - CHANGE: assume conservatively that reinsertion is different
  - CHANGE ◦ CHANGE = ?
    - CHANGE: can’t say any more
  - INSERT ◦ DELETE = ?
    - MISSING (not yet defined)
  - DELETE ◦ DELETE = ?
  - INSERT ◦ INSERT = ?
    - Make no sense
    - Composition is partial
Approach

- **Diff composition**
  - Five operations:
    - CHANGE, INSERT, DELETE
    - UNCHANGED: when a method remains the same in a diff
    - MISSING: when a method is missing from a diff altogether
  - UNCHANGED and MISSING are never produced when diffs are computed, only during composition
  - We include (and distinguish) them because composition is partial
    - e.g. INSERT requires that a method be absent before, and present after
    - INSERT :: Absent → Present
Approach

- Diff composition
  - INSERT :: Absent $\rightarrow$ Present
  - CHANGE :: Present $\rightarrow$ Present
  - DELETE :: Present $\rightarrow$ Absent
  - UNCHANGED :: Present $\rightarrow$ Present
  - MISSING :: Absent $\rightarrow$ Absent
Approach

- Diff composition
  - These ‘types’ tell us that the composition function on diffs has this type:
    \[
    \text{compose} :: (a \to b) \to (b \to c) \to (a \to c)
    \]
  - i.e. consecutive diff operations between versions \(v_1, v_2, v_3\), must agree on the state of \(v_2\)
  - compose is uniquely defined on many inputs
    - compose DELETE MISSING must be DELETE
Approach

- **Diff composition**
  - Ambiguity only arises when selecting between CHANGE and UNCHANGED
    - We’ve not modelled hashes
    - We pick CHANGE conservatively where required

\[
\text{CHANGE} :: \text{Present} \rightarrow \text{Present} \\
\text{UNCHANGED} :: \text{Present} \rightarrow \text{Present}
\]
Approach

- Diff composition
  - compose is associative, but not symmetric:
    - compose \( \text{INSERT} \ \text{DELETE} = \text{MISSING} \)
    - compose \( \text{DELETE} \ \text{INSERT} = \text{CHANGE} \)

```

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>C</th>
<th>D</th>
<th>U</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>⊥</td>
<td>I</td>
<td>M</td>
<td>I</td>
<td>⊥</td>
</tr>
<tr>
<td>C</td>
<td>⊥</td>
<td>C</td>
<td>D</td>
<td>C</td>
<td>⊥</td>
</tr>
<tr>
<td>D</td>
<td>C</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>D</td>
</tr>
<tr>
<td>U</td>
<td>⊥</td>
<td>C</td>
<td>D</td>
<td>U</td>
<td>⊥</td>
</tr>
<tr>
<td>M</td>
<td>I</td>
<td>⊥</td>
<td>⊥</td>
<td>⊥</td>
<td>M</td>
</tr>
</tbody>
</table>
```
## Approach

- **Conflating operations**
  - We can conflate UNCHANGED and MISSING into a single operation UNKNOWN, because they occur in mutually exclusive scenarios.
  - Useful because we implicitly represent them in practice, e.g. when an item is absent.
  - Does not change composition semantics.

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>C</th>
<th>D</th>
<th>UM</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>I</strong></td>
<td>⊥</td>
<td>I</td>
<td>UM</td>
<td>I</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td>⊥</td>
<td>C</td>
<td>D</td>
<td>C</td>
</tr>
<tr>
<td><strong>D</strong></td>
<td>C</td>
<td>⊥</td>
<td>⊥</td>
<td>D</td>
</tr>
<tr>
<td><strong>UM</strong></td>
<td>I</td>
<td>C</td>
<td>D</td>
<td>UM</td>
</tr>
</tbody>
</table>
Approach

- Suggesting upgrades
  - Software composition analysis
  - Given a library with a range of versions,
    - Pick a version which succeeds the current and has no known vulns
    - Compute diff
    - Check if any missing/changed methods are called
    - Make a pull request
Demo

- Update Advisor in SourceClear
Experiments and Evaluation

- SemVer compliance
  - Computed diffs for 114,199 versions across 5,106 libraries from Maven Central, RubyGems, and PyPI
  - 72% of libraries violate SemVer in some version
    - RubyGems: 80%
    - Maven Central: 67%
    - PyPI: 82%
Experiments and Evaluation

● SemVer compliance
  ○ 26% of library versions violate SemVer
    ■ Maven Central: 24%
      ● Raemaekers, et al.
      ● 28.4% to 23.7% over time
    ■ PyPI: 31%
    ■ RubyGems: 31%
Experiments and Evaluation

- SemVer compliance
  - Concrete example: requests
  - Between 2.3.0 and 2.4.0, requests.structures.IteratorProxy was deleted

```python
# requests/structures.py

8  
9   
10  
11  - import os
12   import collections
13  - from itertools import islice
14  -
15  -
16  - class IteratorProxy(object):
17    - """docstring for IteratorProxy"""
18    def __init__(self, it):
```
Experiments and Evaluation

- SemVer compliance
  - Concrete example: requests
  - Between 2.3.0 and 2.4.0, requests.structures.IteratorProxy was deleted
  - Difficult to determine if it was part of public API
    - Python has no access modifiers, only special handling of _ prefix
    - Checked changelogs, commits, documentation
Experiments and Evaluation

- API incompatibilities in open source projects
  - Attempted to perform upgrades automatically on open source projects
  - On average, 10% of upgrades were non-breaking

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>Python</th>
<th>Ruby</th>
</tr>
</thead>
<tbody>
<tr>
<td>Projects</td>
<td>274</td>
<td>422</td>
<td>503</td>
</tr>
<tr>
<td>Direct dependencies</td>
<td>4777</td>
<td>2572</td>
<td>4096</td>
</tr>
<tr>
<td>Direct vulnerable</td>
<td>246</td>
<td>110</td>
<td>250</td>
</tr>
<tr>
<td>Suggested upgrades</td>
<td>150</td>
<td>64</td>
<td>123</td>
</tr>
<tr>
<td>Non-breaking</td>
<td>28 (19%)</td>
<td>0 (0%)</td>
<td>7 (6%)</td>
</tr>
</tbody>
</table>
Threats to validity

- Limitations of static analysis (FP)
  - Call graphs overapproximate dynamic control flow
  - Hashing to detect changes
- Unsupported language features (FN)
- Computing library diffs in isolation (FN)
  - Cannot pick up breaking changes due to calls to methods in transitive upgrades
- Insufficient semantic information (FP)
  - Requests example; must guess if upgrade is really breaking
- Binaries compiled for different platforms (FP)
  - .NET, Java 9
Future work

- Improve false positive and false negative rates
  - Augment static call graphs with dynamic call graphs
  - More sophisticated change detection than hashing
- Upgrade transitive dependencies
  - Find a direct upgrade that performs a transitive upgrade
  - Find the fewest such upgrades
- Handle dependency conflicts
- Suggest better upgrades
  - Constraints, e.g. does not cross a major version
  - Weigh breaking changes vs severity of vulns fixed
- Infer API usage and suggest replacements
Thank you!

Q&A
Try it out

- www.sourceclear.com
- Free trial
- SRCCLR_ENABLE_PR=true SRCCLR_PR_ON=low SRCCLR_IGNORE_CLOSED_PRS=true srcclr scan --url https://github.com/srcclr/example-java-maven --gen-pr
Approach

- Myers’ algorithm

Credit: http://blog.robertelder.org/diff-algorithm/