Detecting Locations in JavaScript Programs Affected by Breaking Library Changes

ANDERS MØLLER, Aarhus University, Denmark
BENJAMIN BARSLEV NIELSEN, Aarhus University, Denmark
MARTIN TOLDAM TORP, Aarhus University, Denmark

JavaScript libraries are widely used and evolve rapidly. When adapting client code to non-backwards compatible changes in libraries, a major challenge is how to locate affected API uses in client code, which is currently a difficult manual task. In this paper we address this challenge by introducing a simple pattern language for expressing API access points and a pattern-matching tool based on lightweight static analysis.

Experimental evaluation on 15 popular npm packages shows that typical breaking changes are easy to express as patterns. Running the static analysis on 265 clients of these packages shows that it is accurate and efficient: it reveals usages of breaking APIs with only 14% false positives and no false negatives, and takes less than a second per client on average. In addition, the analysis is able to report its confidence, which makes it easier to identify the false positives. These results suggest that the approach, despite its simplicity, can reduce the manual effort of the client developers.

CCS Concepts: • **Software and its engineering** → **Software maintenance tools**.

Additional Key Words and Phrases: software evolution, software maintenance, breaking changes

ACM Reference Format:
Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. 2020. Detecting Locations in JavaScript Programs Affected by Breaking Library Changes. *Proc. ACM Program. Lang.* 4, OOPSLA, Article 187 (November 2020), 25 pages. https://doi.org/10.1145/3428255

1 INTRODUCTION

Modern JavaScript applications heavily rely on third-party libraries. The npm registry contains more than 1.2 million packages,[1] and an average package depends on around 80 other packages [Zimmermann et al. 2019]. Many libraries, especially the most widely used ones, are frequently updated. New features are added, security flaws and other bugs are fixed, and outdated functionality is removed. The npm system uses semantic versioning,[2] which is a version numbering scheme that distinguishes between major updates that may contain breaking changes and minor and patch updates that usually can be adapted immediately. For most libraries, a changelog is maintained, documenting the changes, especially the ones that may require attention from the client developers.

Previous work has shown that there is typically a considerable delay, called a technical lag, between a release of a new version of a library and the corresponding update of a client [Zerouali et al. 2019]. Client developers are of course interested in the updates, especially the security-related

[1] http://modulecounts.com/ (April 2020)
[2] http://semver.org/

Authors' addresses: Anders Møller, Aarhus University, Denmark, amoeller@cs.au.dk; Benjamin Barslev Nielsen, Aarhus University, Denmark, barslev@cs.au.dk; Martin Toldam Torp, Aarhus University, Denmark, torp@cs.au.dk.

2475-1421/2020/11-ART187
https://doi.org/10.1145/3428255

ones but also when useful new functionality is added. However, they are often reluctant to switch to the new versions, because of the concern that the updates may break the existing client code. Although serious errors are typically fixed in patch updates, clients may need to update to a new major version of the library; for example, *lodash* version 3.10.1 has a known vulnerability that can only be fixed by updating to (at least) version 4.17.12. This means that it is important for client developers to upgrade and adapt to all the potentially breaking changes in the new version of the library, including those not related to the vulnerability fix.

The problem with the current practice is the high degree of manual effort required. Most importantly, the client developer must consult the changelog and manually identify the relevant places in the client code that need attention. Even with expert knowledge of the client code, this is often a time-consuming and error-prone process. Overlooking a required change may cause working code to fail.

Some library developers provide migration tools, for example the jQuery Migrate Plugin[3] and *lodash-migrate*.[4] Developing and maintaining such specialized migration tools is difficult, which may explain why only a couple of the top 20 most depended upon npm packages[5] come with such tools. These migration tools typically work by injecting runtime warnings when affected library features are encountered in the running client, which means that they require extensive client tests to detect all the relevant places that may need updating. For some libraries, a compatibility layer is provided for major updates, for example *rxjs-compat*,[6] which can be used as temporary workarounds until the client code has been properly adapted. Finally, extensive migration guides are available for some popular libraries, such as *express*, as supplements to the changelogs, but without tool support.

Tools such as *npm audit* and Github automatically notify their users if their code depends on npm module versions that contain known security vulnerabilities. Despite the good intentions of that approach, it is often too coarse-grained to be really useful. Although it may draw attention to the need for updating the client code, it still leaves the burden of finding the relevant parts of the code to the client developer. Moreover, it very often gives false alarms because the vulnerable parts of the library are not being used by the client.

Techniques and tools used for other programming languages are difficult to adapt to JavaScript because of the dynamic nature of the language. As an example, for Java, simply recompiling the client code often reveals which program locations require attention, in the form of static type errors. Specialized tools, such as gofix[7] for Go and Coccinelle [Kang et al. 2019; Padioleau et al. 2008] for C and Java, exploit the existing static type systems of those languages. In comparison, JavaScript does not have a static type system, and static type analysis for JavaScript is notoriously difficult [Kristensen and Møller 2019; Stein et al. 2019].

We propose a semi-automated approach to support JavaScript client developers adapt their code to breaking changes in libraries. The idea is to let the library developer (or someone else familiar with the library) specify the API points that pertain to breaking changes, using a simple pattern-based language. Such a description of breaking change patterns can accompany the usual changelog for each major update of the library. For example, a library developer may express that all calls to method `foo` in module `bar` where the first argument is of type `object` break in version 2.0.0. All clients of the library can then benefit from such a description: We provide a tool that

[3] https://github.com/jquery/jquery-migrate/
[4] https://www.npmjs.com/package/lodash-migrate
[5] https://www.npmjs.com/browse/depended
[6] https://www.npmjs.com/package/rxjs-compat
[7] https://golang.org/cmd/fix/

uses lightweight static analysis to identify all the source locations in the client code that match the patterns and hence may require modifications to adapt to the breaking changes in the library.

Although the breaking change descriptions must be written manually (for now), they are usually quite short and easy to write (as we demonstrate in Section 7), and each library typically has many clients, which makes this modest amount of manual work acceptable. Importantly, most of the breaking changes documented informally in changelogs are expressible in our pattern language. (An example of a breaking change that cannot be captured as a pattern is removing support for outdated JavaScript engines, which generally affects the entire library and typically does not require changes to client code.) In situations where changelogs are unavailable or incomplete, existing tools, such as NoRegrets+ [Møller and Torp 2019], can be used for detecting the breaking changes in the libraries.

We similarly leave performing the actual changes of the client code to the developer; in this paper we focus on how to automate the pattern matching process. The common case is that only a small fraction of the breaking changes in a library update are relevant for a given client, so if not having any tool support, most of the manual effort involved in adapting client code is typically spent on finding the affected pieces of code, not on performing the needed changes. In fact, quite often when a client developer wants to upgrade to a new major version of a library to get access to new functionality, none of the breaking changes affect the client code (see Section 7.2). In that situation, it may take a long time for the client developer to realize that no changes in the client code are needed.

Our key insight is that it is possible to express breaking change patterns in a way that permits accurate and efficient pattern matching based on lightweight static analysis. Ideally, the pattern matching should have neither false positives (reporting locations that are in fact not related to the breaking changes) nor false negatives (missing locations that are related to breaking changes). Achieving that is of course impossible, not only theoretically by Rice's theorem, but also practically due to the known difficulties involved in performing accurate and efficient static analysis for JavaScript, as mentioned above. Some false positives are tolerable, as long as there is not an overwhelming number and they are easy to dismiss manually. It is more important to avoid false negatives; a single false negative may cause a required change to the client code to be overlooked. Existing library-specific migration tools require high-coverage test suites to avoid false negatives, and not many programs have such extensive tests. Our static analysis is efficient and has no false negatives in our experiments (Section 7). Furthermore, the static analysis is designed such that it can report its confidence, which makes it easier to identify false positives. With these properties, the approach is a promising alternative to the current fully manual practice.

In summary, the contributions of this paper are as follows:

- We present a preliminary study of breaking changes in real-world JavaScript packages (Section 3), which has guided the design of our approach.
- We propose a simple pattern language for describing the API access points that are involved in breaking changes, and we provide an accompanying program analysis tool, named TAPIR,[8] for locating which parts (if any) of the client code may be affected by breaking changes (Sections 4–6).
- An experimental evaluation showing that the pattern language is sufficiently expressive in practice, and that the static analysis is accurate and efficient: 187 breaking changes from 15 package updates can be expressed using a total of 283 patterns, and running TAPIR on 265 clients of these packages takes less than a second per client and has a recall of 100% with only 1 in 7 alarms being false positives, and with all high confidence alarms being true positives (Section 7).

[8] **T**ool for **API R**ecognition

2: *Removed category names from module paths*
4: *Removed* `thisArg` *params from most methods because they were largely unused, complicated implementations, & can be tackled with* `_.bind`*,* `Function#bind`*, or arrow functions*
47: *Dropped boolean* `options` *param support in* `_.debounce`*,* `_.mixin`*, &* `_.throttle`
51: *Removed 17 aliases*
Removed `_.all` *in favor of* `_.every`
Removed `_.any` *in favor of* `_.some`
⋮

Fig. 1. The subset of the breaking changes in *lodash* 4.0.0 that affected *postal*. The descriptions are as they appear in *lodash*'s changelog but with examples elided.

While this paper presents a self-contained approach for helping client developers address breaking changes, it can also be viewed as a first step of an automated patching process. We envision a framework where the results from a run of TAPIR is succeeded by a transformation phase that automatically patches the source locations affected by breaking changes. The transformations could be expressed by augmenting the pattern language presented here with some kind of AST transformation language. To ensure correct transformations, the false positive alarms produced by TAPIR have to be removed. However, as false positives mostly belong to the low confidence category, relatively few alarms have to be considered. We do not consider such a larger framework a contribution of this paper, but rather a direction for future work.

2 MOTIVATING EXAMPLE

Let us consider the npm package *postal*, an in-memory message bus library, which is currently downloaded more than 20 000 times weekly. Based on its git commit history we can reconstruct a typical update scenario. On April 30, 2016 the maintainer of *postal* decided to update the *lodash* dependency from version 3.10.1 to 4.11.1, which was the newest version at the time. The *postal* maintainer was aware of a breaking change in *lodash*'s debounce method. He therefore located the places in *postal*'s source code where debounce was used and updated the code accordingly. He then pushed a patch update of *postal* to the npm registry, probably assuming that no other breaking changes in *lodash* were affecting *postal*.

Later that same day, however, the *postal* maintainer discovered that *postal* was affected by yet another couple of breaking changes introduced in the update of *lodash*. He probably learned this by observing that *postal*'s test cases no longer succeeded. He found four places in the source code that were affected by these changes, patched the code, and pushed a new patch update of *postal* to the npm registry.

A few weeks later, a user of *postal* discovered yet another set of breaking changes affecting *postal* and created a pull request with the required fixes. These changes were not caught by the test suite of *postal*, which is probably why they were not found sooner. Two weeks later, the *postal* maintainer finally merged the pull request and created a new version of *postal* fully adapted to the new version of *lodash*.

Consequently, the newest version of *postal* in the npm registry for more than a month was not properly adapted to work with *lodash* in the version 4 major range, which could lead to crashes and misbehavior of clients depending on *postal*. This example clearly illustrates the difficulties in adopting major updates of dependencies. In essence, the client maintainer must first go through the list of all documented breaking changes in the update, then determine which breaking changes are relevant for the client, and then adapt the client code to the breaking changes. As the *postal* example

```
Detection pattern 4 matched at lib/postal.js:596:8 with low confidence
Detection pattern 47 matched at lib/postal.js:250:12 with low confidence
Detection pattern 51 matched at lib/postal.js:49:45 with low confidence
Detection pattern 51 matched at lib/postal.js:109:26 with low confidence
Detection pattern 51 matched at lib/postal.js:582:108 with low confidence
Detection pattern 2 matched at lib/postal.lodash.js:14:11 with high confidence
Detection pattern 2 matched at lib/postal.lodash.js:19:14 with high confidence
Detection pattern 2 matched at lib/postal.lodash.js:26:13 with high confidence
Detection pattern 2 matched at lib/postal.lodash.js:27:10 with high confidence
Detection pattern 2 matched at lib/postal.lodash.js:29:14 with high confidence
Detection pattern 47 matched at lib/postal.lodash.js:273:12 with low confidence
```

Fig. 2. The output of TAPIR after analysis of the source code of *postal* using the breaking change detection patterns for *lodash* 4.0.0.

illustrates, relying on test suites will not always catch all the client code locations that are affected by breaking changes. The test suites of the clients are designed to test the client code, not to check for failures in dependencies, so even high-quality test suites can be inadequate for this purpose. The client developer usually has no other option than reading through the changelog of the dependency and then manually determining which breaking changes are relevant for the client's usage of the dependency. This is a time consuming and error-prone task since many clients use only a small subset of the APIs of their dependencies, so often only a few or none of the breaking changes are actually relevant for each client. For example, *postal* is only affected by 4 of the 54 breaking changes introduced in *lodash* version 4.0.0. The relevant items from *lodash*'s changelog are shown in Figure 1 (see the full list on https://brics.dk/tapir/). Evidently, finding these relevant items among all the 54 entries in the changelog is a major effort if done manually, even for someone deeply familiar with the *postal* source code. Interestingly, the second one (about removed `thisArg` params) is relevant for *postal* despite being described as "largely unused" in the changelog.

Using TAPIR, the push of a single button will list all of the 9 places in *postal*'s source that had to be changed when updating the *lodash* dependency, due to breaking changes in the library API. As we explain in Section 7, the breaking changes in *lodash* version 4.0.0 can be concisely captured by a collection of patterns that describe the affected API access points, making it substantially easier to update all the many thousands of clients of *lodash*. The actual output of the tool is shown in Figure 2. A manual inspection reveals that 2 of the 11 matches reported are false positives (meaning that those locations in the program are actually not affected by any of the breaking changes). As part of its output, TAPIR shows its confidence, and those two cases are indeed matches with low confidence. All the other locations are true positives that require small changes to adapt to the new version of the library. Conversely, all the changes made manually by the *postal* developer to adapt to *lodash* 4.0.0 are correctly detected by TAPIR (disregarding a couple of places where *postal* accesses internal functionality of *lodash* that is not part of its public API and therefore not mentioned in the changelog).

In conclusion, if TAPIR had been available to the *postal* maintainer, it would likely have substantially reduced the manual workload, and it would likely also have prevented the broken *postal* version from ever reaching the npm registry.

3 PRELIMINARY STUDY

To understand what kinds of breaking changes package maintainers typically introduce in major updates, we have conducted a manual study of the changelogs from 10 major updates of some of

Table 1. Breaking changes in npm packages.

	Module		**Property**		**Function**				
Library	**Removal**	**Move**	**Removal**	**Move**	**Signature**	**Behavioral**	**Env.**	**Build**	**Total**
lodash 4.0.0	0	3	4	13	24	7	3	0	54
async 3.0.0	0	0	0	1	2	1	1	0	5
express 4.0.0	0	0	9	2	1	4	2	1	19
chalk 2.0.0	0	0	3	0	0	0	1	0	4
bluebird 3.0.0	0	0	2	0	3	2	0	0	7
uuid 3.0.0	0	0	1	0	0	0	0	0	1
commander 3.0.0	0	0	0	0	0	3	0	1	4
rxjs 6.0.0	0	13	2	1	0	2	6	6	30
core-js 3.0.0	3	8	11	2	0	2	2	0	28
yargs 14.0.0	0	0	0	0	0	1	0	0	1
Total	**3**	**24**	**32**	**19**	**30**	**22**	**15**	**8**	**153**

the most widely used npm packages. As a methodology for selecting packages, we picked the top 10 packages with the highest number of direct dependents in the npm registry, disregarding a package if its newest version was less than 1.0.0, or if the changelog was unavailable or did not mention the latest major version. As we focus on the Node.js platform, we also chose to disregard packages that are used only for front-end web development or in build systems (e.g., *react* and *webpack*).

Each of the changelogs contains a bullet list of changes. We disregarded changes that do not break backward compatibility, including addition of new features and bug fixes. (In theory, clients may apply workarounds for known bugs, which may cause the client code to break when the bug is fixed, but we ignore that here.) Each changelog bullet is counted as one change, even though one single bullet sometimes covers many library functions. (For example, the changelog of *lodash* 4.0.0 contains a single bullet describing the removal of 17 aliases; see breaking change number 51 in Figure 1.) Furthermore, we disregarded changes explicitly marked as deprecations. (For such changes, the old behavior or feature is still present, but new clients are discouraged from using it, and in many cases the deprecated features are scheduled to be removed in some future major update, in which case they will then be treated as actual breaking changes.)

Interestingly, the changelogs do not always clearly specify which parts of the API are involved in a change. (An example from *lodash* is breaking change number 4 shown in Figure 1; a closer inspection reveals that this one affects 64 different functions.) This means that even for manual use by the client developers, the existing informal changelogs do not provide enough information to be able to safely adapt the client code.

To learn about the nature of the collected real-world breaking changes, we grouped them into a number of categories. The collection of packages and the number of changes belonging to each category are listed in Table 1.

First, we have three primary categories (**Module**, **Property**, and **Function**) concerning changes that are related to specific points in the package APIs. A **Module** change is one where an entire module is either removed completely (for example, the *core-js/client/library* module is removed in *core-js* version 3.0.0) or moved to a different location (for example, *core-js/library/fn/parse-int* is moved to *core-js/features/parse-int*, also in *core-js* version 3.0.0), which we show as two different sub-categories. A **Property** change is one where a property (typically a method) of an object is removed or moved to another object. The property removal category includes a few cases where client-written properties are no longer read by the library. (For example, prior to version 3.0.0 of *async*, a client could write functions to the `drain` and `saturated` properties on special queue objects, which would then be called at specific events. In version 3.0.0, `drain` and `saturated` are instead functions that must be called with the event handler functions as arguments.) A **Function** change is one where the signature or behavior of a function has been modified. Function signature changes (**Signature**) include reordering, removals, and addition of parameters, and changes to parameter types or return types, but also cases where a function is conditionally renamed based on the arguments. (For example, in *lodash* version 4.0.0 clients must use `sumBy` instead of `sum` if the client supplies an optional function argument, which is sometimes used to specify how to sum over each element.) Behavioral changes (**Behavioral**) are all changes to functions that do not affect the function signatures but modify the semantics. (For example, in the update of *lodash* to version 4.0.0, the `functions` function, which previously returned all function properties of its argument, was changed to no longer include inherited properties.)

The changes in the remaining categories (**Env.** and **Build**) do not relate to specific points in a package API but to a package as a whole. The **Env.** category contains changes that only affect specific execution environments, such as removal of support for Internet Explorer 7 or outdated versions of Node.js. The **Build** category consists of changes that may affect the build process of client applications, for example, causing the compilation of TypeScript-based clients to fail. From the client developer's perspective, these categories of breaking changes are quite different from the other ones. For example, the client developer can probably decide whether or not it is acceptable to lose support for old platforms without needing any source code changes, and tooling to detect TypeScript compilation errors is already available.

This study has several interesting findings, which we leverage in the design of our solution in Section 4.

Most importantly, we see from Table 1 that the majority of breaking changes, 130 out of 153, belong to the categories concerning specific points in the package APIs (**Module**, **Property**, and **Function**).

For the **Module** changes, it is particularly easy to find the affected places in the client code, simply by searching for locations where the module in question is being loaded.

The **Property** changes are potentially more challenging. Due to the dynamic nature of JavaScript, it is generally difficult to statically track the flow of objects that originate from the package in question, to be able to find the places in the client code that may be affected by the breaking changes. This is the main challenge we address in the following section. As an example, consider the breaking change in the reactive-programming library *rxjs* version 6.0.0 update, where chainable operators are removed from the special observable type. Objects of the observable type can be created in many different ways, not only using various constructors, but also as results of operations on other observables. Furthermore, observables are commonly passed around in client code, which makes it difficult to determine which variables refer to observables. If a client function contains the expression `x.map(...).filter(...)` where `x` is some argument to that function, it is hard to statically determine whether `x` is an observable or just an ordinary JavaScript array. We thus need a mechanism for addressing the affected parts of the package API, and a mechanism for automatically pointing the client developer to the locations in the client code that use those parts of the API.

Functions in package APIs are accessed by clients in the same way as other properties, so the changes in the **Function** category can benefit from the same mechanisms. However, it may be beneficial to provide extra precision, to be able to report only the functions that are called with specific types or numbers of arguments. For example, for the `sum` function mentioned above, the breaking change is only relevant when `sum` is called with two arguments. As another example, a breaking change in *lodash* version 4.0.0 affects the methods `debounce`, `mixin`, and `throttle` only if the third argument is of type boolean. Sometimes, even behavioral changes may benefit from such a filtering mechanism. For example, in *commander* version 3.0.0, there is a behavioral change only affecting calls to the `on` method where the first argument is a string and the second is a function.

```
            Pattern ::= import(D)? Glob
                      | read PropertyPathPattern
                      | write PropertyPathPattern
                      | call(R)? AccessPathPattern (Filter)*

               Glob ::= (GlobElement)*

        GlobElement ::= Filename | * | /**/ | /
                      | { Glob , ... , Glob }

  AccessPathPattern ::= < Glob >
                      | { AccessPathPattern , ... , AccessPathPattern }
                      | ( AccessPathPattern \ AccessPathPattern )
                      | AccessPathPattern ()
                      | AccessPathPattern **
                      | PropertyPathPattern

PropertyPathPattern ::= AccessPathPattern . Property
                      | AccessPathPattern .{ Property , ... , Property }

             Filter ::= [ Int , Int ] | [ Int ,]
                      | Int : Type | Int : { Type , ... , Type }

               Type ::= string | number | boolean
                      | undefined | object | array | function
                      | function[ Int ] | Literal
```

Fig. 3. Pattern language for describing API access points.

4 THE TAPIR APPROACH

We have developed the tool TAPIR for finding the source locations in client code that may be affected by the breaking changes in a major update of a dependency. In Section 5 we introduce a simple pattern language for specifying API access points where breaking changes occur and what conditions must be met for the breaking changes to be relevant.

We envision that the library developer writes patterns in this language while developing a major update, to accompany the changelogs that are traditionally written. However, as we demonstrate in Section 7, a collection of patterns for a library update is typically relatively small and quite easy to write, even without expert knowledge of the library, so it is also possible that, for example, a client developer performs this task. This resembles the practice for TypeScript declaration files,[9] which are also often contributed to the community by other programmers than the JavaScript library developers themselves. In situations where comprehensive changelogs are not available, existing tools can be used for detecting the breaking changes in the libraries [Møller and Torp 2019].

Once a collection of patterns has been written for a library update, it can be reused for all the clients that need to be adapted. Especially for widely used libraries with thousands or millions of clients, this justifies the effort required to write the patterns. The client developers can use the TAPIR tool, which applies a lightweight static analysis to the client code as explained in Section 6, to automatically find the places in the client code that are affected by the specified breaking changes.

5 A PATTERN LANGUAGE FOR DESCRIBING API ACCESS POINTS

Figure 3 shows the syntax of our pattern language for describing API access points of interest. There are four kinds of patterns.

[9] https://github.com/DefinitelyTyped/DefinitelyTyped

First, an *import* pattern, written `import` q where q is a Unix path-like pattern (also known as a glob pattern), matches client code that imports a module with a name described by q. Such patterns are intended for describing the **Module** breaking changes from Section 3. A glob consists of a file system path that, when matched against a concrete file system, results in a set of matched files or directories. A glob can also contain wildcards, such as the `*` that matches all files, `**` that matches all directories recursively, and $\{i_1, \ldots, i_n\}$ that matches the union of the globs i_1 to i_n. The specialized import pattern `importD` is described at the end of this section.

Example 1. The pattern

```
import core-js/client/*
```

matches both `require('core-js/client/library')` and `import * as lib from 'core-js/client/core'`, and also other ways of loading modules with names that begin with `core-js/client/`.

A *read* pattern, written `read` p, matches property read operations in the client code that match the property path pattern p. Most of the **Property** changes from Section 3 can be described by read patterns. A *write* pattern, `write` p, similarly matches property write operations. We introduce a notion of access path patterns to be able to characterize the relevant dataflows:

- $<q>$ matches the same client code as the import pattern `import` q.
- $\{\, p_1 \,,\, \ldots \,,\, p_n \,\}$ matches the union of the expressions matched by the sub-patterns $p_1, \ldots, p_n$.
- $(\, p \setminus p' \,)$ matches everything that matches p and not p'.
- $p()$ matches function (and method) call expressions (with or without `new`) where p recursively matches the sub-expression that provides the function (or method). Intuitively, the pattern describes the return value of the call.
- p`**` matches all chains of property reads and method calls on expressions matched by p.

Additionally we have two kinds of access path patterns called property path patterns:

- $p.f$ matches all property read and write operations $E.f$ in the client code (for a JavaScript expression E and a property name f) where p recursively matches E. (In principle this kind of pattern also matches dynamic property accesses where the property name is dynamically computed and may evaluate to f, but see Section 6.)
- $p.\{f_1, \ldots, f_n\}$ is similar to the preceding kind but matches any of the given property names.

In addition to these rules, if a pattern p matches an expression E_1 in the client code and the run-time value of E_1 may flow (via assignments, function calls, etc.) to another expression E_2, then p also matches E_2. In Section 6 we show how to approximate this statically using a simple alias analysis.

Example 2. The following read pattern describes a breaking change in the *lodash* version 4.0.0 update.

```
read <lodash>.any
```

This pattern matches all reads of the `any` property on the *lodash* module, which was moved in the update.

Example 3. The following write pattern describes the breaking change in the *async* version 3.0.0 update that was mentioned in Section 3.

```
write <async>.queue().{drain,saturated}
```

It matches writes to the `drain` and `saturated` properties on objects created by calling the `queue` function on the *async* module.

A *call* pattern, `call` $p\ f_1\ \ldots\ f_n$, matches function/method/constructor call operations that match p and also satisfy each of the filters $f_1, \ldots, f_n$. The call patterns are designed to handle the **Function** changes from Section 3. We use a notion of filters to describe the changes that are conditional on the number of arguments or the argument types.

- `[`n`,`m`]` restricts the matching to calls with between n and m arguments.
- `[`n`,]` restricts the matching to calls with at least n arguments.
- n`:`t restricts the matching to calls where the n'th argument has type t.
- n`:{`t_1`, ... ,`t_n`}` is variant of the preceding kind that permits a union of types $t_1, \ldots, t_n$.

The types we support for filtering include the usual JavaScript types (string, number, boolean, undefined, object, array, function), but we also allow singleton types (expressed as *Literal*) and functions with specific numbers of parameters (`function[`*Int*`]`), which are useful for describing callbacks. Another task of the static analysis presented in Section 6 is to approximate this filtering information statically for a given client and pattern. The specialized call pattern `callR` is described at the end of this section.

Example 4. The following call pattern describes the `thisArg`-related breaking change in the *lodash* version 4.0.0 update mentioned in Section 3.

```
call <lodash>.each [3,3]
```

The pattern matches all calls to `each` on the *lodash* module, where `each` is called with exactly 3 arguments. The '`thisArg`' is an optional third argument, so if `each` is called with fewer than 3 arguments, then the call is not affected by the breaking change. In this case, no constraints on the argument types are required since the breaking change is relevant for all `each` calls with 3 arguments.

Example 5. The version 3.0.0 update of the command-line parser utility *commander* contains a breaking change affecting the `parse` method. That method is typically called as the last method in a long chain of method calls, so we can recognize it as follows using `**`:

```
call <commander>**.parse
```

Example 6. The following pattern recognizes the removal of the `merge` observable creator function from observable objects.

```
read (<rxjs>** \ <rxjs>.Observable).merge
```

The `merge` method still exists on the object denoted by the `Observable` property, so the pattern must ensure that reads of `merge` on this object are not matched. This constraint is enforced by using the \ operator to require that `<rxjs>**` matches but `<rxjs>.Observable` does not.

The `call` and `import` patterns have specialized forms, which are sometimes useful to improve precision. The specialized `call` pattern, written `callR`, will only match a call if the return value of that call is not discarded. The specialized `import` pattern, written `importD`, will only match imports that use the default import mechanism. Both constraints are easily checked by considering the syntax around calls and imports in the client code.

Example 7. Consider the following breaking change affecting the MongoDB object modelling tool *mongoose* when updated to version 5. The `connect` method of *mongoose* returns an object of type `MoongooseThenable` in version 4. In version 5 a built-in JavaScript promise, which has a slightly different API, is instead returned. Because we observed that many clients ignore the return value of `connect`, and this breaking change is only relevant when the return value is somehow used, the following pattern is preferable to using an ordinary `call` pattern:

```
callR <mongoose>.connect
```

Example 8. The *rxjs* library introduces a breaking change in the update to version 6, where default imports from the `'rxjs'` module are no longer supported. Prior to version 6, clients could write `import rx from 'rxjs'` to load what is known as the default exported object into the `rx` variable.

However, the developers of *rxjs* wanted to encourage clients to only load the functions from `'rxjs'` that are used in the code. For example, clients should instead write `import {merge,interval} from 'rxjs'` if the `merge` and `interval` methods are used. Therefore, the ability to use a default import was removed from the module. To catch this breaking change we use the pattern

```
importD rxjs
```

which matches only those imports from `'rxjs'` that import the default exported object. Thereby, clients who already load only the required methods from the module will not get any warnings.

6 A STATIC ANALYSIS FOR FINDING USES OF API ACCESS POINTS IN CLIENT CODE

To find out which parts of the client code may be affected by breaking changes in the library, TAPIR scans the client code for expressions that match one of the API access point patterns. For this purpose, TAPIR uses a lightweight AST-based static analysis that is designed to be fast and achieve a high recall (to minimize the number of false negatives) while sacrificing some precision (allowing a modest number of false positives).

The static analysis of TAPIR is separated into three phases that are run in isolation on each file of the client code. The first phase is an alias analysis that finds expressions that may alias a given variable or object property. The second phase infers access paths for all expressions, to identify the connections between the client code and the imported modules. The third phase performs pattern matching, to find the expressions that have an access path that matches one of the API access point patterns of interest. For each match, the user is notified with the source location of the expression, as shown in Figure 2.

6.1 Phase 1: Alias Analysis

The alias analysis is a flow-insensitive, field-based[10] analysis that computes a map

$$\alpha_s : \mathit{Var} \cup \mathit{Prop} \rightarrow \mathcal{P}(\mathit{Exp})$$

for each source file s, where Var, Prop, and Exp are, respectively, the set of declared variables (including function parameters), the set of property names, and the set of expressions that occur in the current source file. For a variable $x \in \mathit{Var}$, $\alpha_s(x)$ approximates the set of expressions that may alias x (meaning that x and the expression may evaluate to the same object at run-time). Similarly, for a property name $f \in \mathit{Prop}$, $\alpha_s(f)$ approximates the set of expressions that may alias the f property of some object.

The alias analysis is extremely simple: The map is constructed in a single scan through the AST of the source file. At each assignment[11] $x = E$ or $E'.f = E$, the expression E is simply added to $\alpha_s(x)$ or $\alpha_s(f)$, respectively.

Example 9. For a chain of assignments such as `x = require('lib')` and `y = x`, the analysis infers $\alpha_s(\texttt{x}) = \{\texttt{require('lib')}\}$ and $\alpha_s(\texttt{y}) = \{\texttt{x}\}$; it does not model transitive dataflow, so $\alpha_s(\texttt{y})$ does not contain `require('lib')`. This may seem like a serious weakness for an alias analysis, but we compensate when the alias analysis results are being used in the second phase, which we explain later.

The analysis completely ignores dynamic property write assignments, the flow of objects at function calls and returns, exceptions, etc. The analysis design is inspired by Feldthaus et al. [2013] who found field-based analysis to be highly scalable and surprisingly accurate; in particular, ignoring dynamic property accesses in their analysis loses only little soundness in practice. Unlike their

[10] A field-based analysis abstracts heap locations only by the property names, without distinguishing between objects.

[11] Each import, such as `import {map} from 'lib'`, is treated as an assignment, `var map = require('lib').map`.

analysis, we additionally ignore function calls and returns, which of course makes the analysis even more unsound in theory, but we find that typical JavaScript code rarely passes module objects through function calls or returns. (We discuss in Section 8 why we are not using the call graph construction of Feldthaus et al.)

Also note that each source file is analyzed separately. This is reasonable since JavaScript's module system uses one file per module, so interactions between files take place via the module import mechanism.

Example 10. Consider the following example that asynchronously adds line numbers to the beginning of the files `'file1'` and `'file2'`, and then outputs the total number of line numbers added.

```
 1 const _ = require('lodash');
 2 const libAsync = require('async');
 3 const fs = require('fs').promises;
 4
 5 (async function () {
 6   const lines = await libAsync.map(['file1', 'file2'],
 7     async (file) => {
 8       const lns = (await fs.readFile(file)).split('\n');
 9       const idxLns = _.map(lns, (ln, idx) => idx + ':' + ln);
10       await fs.writeFile(file, idxLns.join('\n'));
11       return lns.length;
12     });
13   console.log('Added ' + _.sum(lines) + ' line numbers');
14 })();
```

The example uses the `map` function from the *async* library in line 6 for asynchronously applying the transformation to both files. The insertion of the line numbers is done in line 9 using another function named `map`, which comes from the *lodash* library. Assume (hypothetically) that *lodash*'s `map` method is removed in a major update (this would obviously be a breaking change). In this situation, we would like TAPIR to identify all source locations where the `map` property from *lodash* is read. This change is captured by the following pattern:

```
read <lodash>.map
```

The alias analysis provides the information needed to avoid confusing the call in line 6 with a call to *lodash*'s map function, since it knows that the value of the _ variable is the *lodash* module and that the value of the `libAsync` variable is the *async* module:

$$\begin{aligned} \alpha_s : [\; & \texttt{\_} \mapsto \{\texttt{require('lodash')}\}, \\ & \texttt{libAsync} \mapsto \{\texttt{require('async')}\}, \\ & \ldots] \end{aligned}$$

6.2 Phase 2: Access Path Inference

The alias information is used in the second phase of TAPIR to establish the connections between the client code and the imported modules. In this phase, TAPIR computes a set of *access paths* for each import, call, read property, and write property expression in the client program. The structure of access paths is defined by the following grammar:

$$\begin{array}{lcl} \mathit{AccessPath} & ::= & <\mathit{ImportPath}> \\ & | & \mathit{AccessPath}\ .\ \mathit{Property} \\ & | & \mathit{AccessPath}\ () \\ & | & \mathtt{U} \end{array}$$

The first three productions model module imports, property reads, and function calls, respectively. The symbol U models unknown access paths.

$$AP_S(E) \;:=\; \begin{cases} \{\texttt{<}m\texttt{>}\} & \text{if } E = \texttt{require}(m) \\ \{ap\,.f \mid ap \in AP_S(E')\} \,\cup\, lookup_S(f) & \text{if } E = E'.f \\ \{ap\texttt{()} \mid ap \in AP_S(E')\} & \text{if } E = E'\texttt{(...)} \text{ or } E = \textbf{new}\; E'\texttt{(...)} \\ lookup_S(x) & \text{if } E = x \text{ where } x \text{ is a variable that is not a parameter} \\ \{\mathsf{U}\} & \text{otherwise} \end{cases}$$

$$lookup_S(z) \;:=\; \begin{cases} \bigcup_{E \in \alpha_s(z)} AP_{S \cup \{z\}}(E) & \text{if } z \notin S \\ \emptyset & \text{otherwise} \end{cases}$$

where z is a variable (excluding parameters) or a property name,
and s is the current source file

Fig. 4. Access path inference.

The function AP for computing access paths for a given expression E is defined in Figure 4.

- If E is a module load operation, such as `require('lodash')`, AP returns the corresponding import path.
- For a property read $E'.f$, the access paths are computed by recursively computing the access paths of E' and adding $.f$, and by recursively calling AP on all aliased expressions using the *lookup* function, which uses the alias map α_s from phase 1. Notice that *lookup* calls AP recursively and thereby takes care of transitive dataflow as discussed in Example 9; see also Example 12 below.
- For a function call expression E'`(...)` (with or without `new`), AP similarly constructs the access paths by recursively computing the access paths for E' and then appending `()`.
- When E is a variable read (excluding function parameters), the access paths are obtained using the *lookup* function.
- Otherwise, AP returns $\{\mathsf{U}\}$ to indicate that we do not have any knowledge about the access paths for the expression. This case covers, for example, function parameters, arithmetic operators, and literals.

The subscript S in the recursive definition of AP is used for ensuring termination in case of recurrences of expressions, as illustrated by Example 13 below. When we write AP without the subscript, it is implicitly the empty set.

Example 11. Continuing Example 10, we can now compute the access paths using the definition of AP. The set of access paths for the `map` property read on line 6 computes to the singleton set {`<async>.map`}, and the access paths for the `map` read on line 9 computes to the set {`<lodash>.map`}.

Example 12. As a variant of Example 9, consider the following code.

```
function (x) {
  x.f = require('lib').fun;
  x.f(42);
}
```

The result of the alias analysis contains α_s(`f`) = {`require('lib').fun`} (assuming there are no other assignments to the `f` property of some object). In phase 2, the set of access paths for `x.f` at the call statement is computed to AP(`x.f`) = {`<lib>.fun`, `U.f`}. The access path `U.f` appears spuriously because the analysis is flow-insensitive and ignores parameter passing at calls.

Example 13. For the following code,

```
var x = require('lib');
x = x.f;
```

computing $AP(\texttt{x.f})$ leads to a recurrence of `x`, so the resulting set of access paths is $\{\texttt{<lib>.f}\}$ (assuming this is the only code being analyzed). To see this, first notice that the alias analysis gives $\alpha_s(\texttt{x}) = \{\texttt{require('lib')}, \texttt{x.f}\}$ and $\alpha_s(\texttt{f}) = \emptyset$. According to the definition of AP, we then have

$$\begin{aligned}
AP_{\emptyset}(\texttt{x.f}) &= \{ap.\texttt{f} \mid ap \in AP_{\emptyset}(\texttt{x})\} \;\cup\; \mathit{lookup}_{\emptyset}(\texttt{f}) \\
&= \{ap.\texttt{f} \mid ap \in \mathit{lookup}_{\emptyset}(\texttt{x})\} \;\cup \bigcup_{E \in \alpha_s(\texttt{f})} AP_{\{f\}}(E) \\
&= \{ap.\texttt{f} \mid ap \in \bigcup_{E \in \alpha_s(\texttt{x})} AP_{\{\texttt{x}\}}(E))\} \\
&= \{\texttt{<lib>.f}\}
\end{aligned}$$

since

$$AP_{\{\texttt{x}\}}(\texttt{require('lib')}) = \{\texttt{<lib>}\}$$

and

$$\begin{aligned}
AP_{\{\texttt{x}\}}(\texttt{x.f}) &= \{ap.\texttt{f} \mid ap \in AP_{\{\texttt{x}\}}(\texttt{x})\} \;\cup\; \mathit{lookup}_{\{\texttt{x}\}}(\texttt{f}) \\
&= \{ap.\texttt{f} \mid ap \in \mathit{lookup}_{\{\texttt{x}\}}(\texttt{x})\} \;\cup \bigcup_{E \in \alpha_s(\texttt{f})} AP_{\{f\}}(E) \\
&= \emptyset.
\end{aligned}$$

As mentioned earlier, the alias analysis does not track interprocedural dataflow. In most cases, this limitation is not a practical problem, because typical library usage is of a relatively simple form where a module is loaded, stored in some variable, and then the usage of the library comes from calling methods on this variable. In these cases, the module object structure is essentially used as a static namespace mechanism. To support the remaining more dynamic cases where library objects are being passed to and from functions, we allow a relaxed form of patterns, written $p?$, which matches the same access paths as p but conservatively also `U`.

Example 14. The API access path pattern

```
write <async>.queue()?.{drain,saturated}
```

matches both the access paths

```
<async>.queue().drain
```

and

```
U.saturated
```

but not

```
<lodash>.queue().drain
```

since the latter is clearly not related to the *async* module.

Based on our experience, it is generally quite easy to decide whether to use p or $p?$. One can always choose the $p?$ variant if in doubt since it overapproximates the other one, but it may result in more false positives in the pattern matching in phase 3. However, as we show in the evaluation, the number of false positives is not as problematic as one might expect, since variable and property names tend not to overlap much across different libraries. Overall, this design choice of introducing the relaxed form of detection patterns is a pragmatic compromise between simplicity of the pattern language and scalability and accuracy of the analysis; we return to this discussion in Section 8.

$$\overset{\textsc{import}}{\frac{m \text{ matches } q}{\texttt{<}q\texttt{>} \succ \texttt{<}m\texttt{>}}} \qquad \overset{\textsc{uncertain-1}}{\frac{p \succ t}{p? \succ t}} \qquad \overset{\textsc{uncertain-2}}{\frac{}{p? \succ \mathsf{U}}}$$

$$\overset{\textsc{disjunction}}{\frac{p_i \succ t \qquad i \in \{1, \ldots, n\}}{\{p_1, p_2, \ldots, p_n\} \succ t}} \qquad \overset{\textsc{exclusion}}{\frac{p \succ t \qquad p' \not\succ t}{p \backslash p' \succ t}}$$

$$\overset{\textsc{star-1}}{\frac{p{*}{*} \succ t}{p{*}{*} \succ t.f}} \qquad \overset{\textsc{star-2}}{\frac{p{*}{*} \succ t}{p{*}{*} \succ t()}} \qquad \overset{\textsc{star-3}}{\frac{p \succ t}{p{*}{*} \succ t}}$$

$$\overset{\textsc{prop-read-1}}{\frac{p \succ t}{p.f \succ t.f}} \qquad \overset{\textsc{prop-read-2}}{\frac{p \succ t \qquad f \in \{f_1, \ldots, f_n\}}{p.\{f_1, \ldots, f_n\} \succ t.f}} \qquad \overset{\textsc{call}}{\frac{p \succ t}{p() \succ t()}}$$

Fig. 5. Pattern matching relation $p \succ t$ where p is an access path pattern and t is an access path.

6.3 Phase 3: Pattern Matching

The third phase of the TAPIR analysis matches the API access point patterns against the client code. The matching is defined as a relation $\succ$ between access path patterns and access paths as seen in Figure 5. The inference rules directly reflect the descriptions of the access path patterns from Section 5 (the only exception being the two UNCERTAIN rules for handling $p?$ as discussed in Section 6.2).

For an `import` pattern, performing the matching is straightforward. The analysis matches the access paths computed for every module load AST node in the previous phase against the import pattern using the IMPORT rule (as shown in Example 1).[12] For an `importD` pattern, it must furthermore be the case that the default loading mechanism is used, which is always trivially decidable from the syntax of the module load.

For `read` and `write` patterns, the read and write operations in the client code are matched against the access paths computed by the previous phase of TAPIR using the $\succ$ relation.

Example 15. Consider the JavaScript code and the `read` pattern from Example 10. Based on the access paths computed for this client code (see Example 11), TAPIR finds a match on line 9, but not on line 6.

For a `call` pattern, TAPIR similarly looks at every call node in the client code. First it matches the access paths of the function (computed in the previous phase) against the access path pattern using $\succ$ exactly as with `call` and `write` patterns. For all calls where an access path matches, TAPIR matches the arguments of the call against the filters of the pattern. The number of arguments to the call can be extracted directly from the AST.[13] For type filters, TAPIR also attempts to extract the argument type directly from the argument AST node: If the argument is a literal, then the type is simply the type of that literal; otherwise, TAPIR conservatively assumes that the type filter matches

[12] TAPIR does not support dynamically computed strings, but they are rarely used for loading modules in practice.

[13] For calls made using `Function.prototype.apply` or using the spread operator, TAPIR conservatively assumes that the `[`n`,`m`]` filter always matches.

(but with low confidence; see Section 6.4). For a `callR` pattern to match, the result of the call must not be discarded.

Example 16. Consider the pattern from Example 4 and this modified excerpt from the *postal* application.

```
21 var _ = require('lodash');
22 ...
23 _.each(_.keys(this.cache), function (cacheKey) {
24   ...
25 }, this);
```

The access paths of `_.each` on line 23 are computed as the singleton set {`<lodash>.each`} whose single entry matches the access path pattern. Therefore, TAPIR checks the `[3,3]` filter against the arguments of the call, which in this case results in a match.

Example 17. Had the call on line 23 instead been a call to the `dropWhile` function (which is also affected by the `thisArg` breaking change), then a type filter would also be required in the pattern:

```
call <lodash>.dropWhile [3,3] 2:function
```

The *lodash* library supports a shorthand syntax for the `dropWhile` function. As an example, one can write `dropWhile(x,'foo',42)` instead of `dropWhile(x,(o) => o.foo === 42)`. The shorthand variant also takes three arguments but is not affected by the breaking change, so the pattern has to be extended to also check that the second argument is of type `function`.

6.4 Reporting Analysis Confidence

As a convenient extra feature of the static analysis, it can classify each match as either *high confidence* or *low confidence*, as shown in the example output in Figure 2. The intention is that this extra information can be useful for the client developer when deciding how to adapt the code to the breaking changes in the library. The client developer can trust the high confidence matches and only needs to manually review the low confidence matches, thereby further reducing the manual work. (Of course, this requires that high confidence matches are not false positives in practice; we demonstrate experimentally in Section 7 that the confidence classification has this property.)

As explained in Section 6, the analysis is designed such that it is unlikely to have false negatives, but it may have false positives meaning that it can report spurious matches. Despite the simplicity of the analysis, there are only three sources of likely false positives: the relaxed form of patterns p? (see Section 6.2), the filters at call patterns (see Example 17), and inference of multiple access paths for an expression. This gives us a simple confidence classification mechanism. A match between a pattern p and an expression E is classified as low confidence if

- p has one or more occurrences of ?, and it no longer matches E if removing them,
- p contains one or more call filters, and the pattern matcher cannot trivially determine that they match the corresponding arguments in the AST, or
- multiple access paths are inferred for E and not all of them match p,

and otherwise it is a high confidence match.

The following examples illustrate each of the three conditions for low confidence.

Example 18. Assume p is the pattern `<async>.queue()?.{drain,saturated}` from Example 14 and E is the property write expression `x.drain = ...` in this function:

```
26 function f(x) {
27   ...
28   x.drain = ...
29   ...
30 }
```

In this case, the analysis is unable to determine with certainty whether `x` matches `<async>.queue()`. The single access path `U.drain` is inferred for `x.drain`, and by the pattern matching mechanism we have `<async>.queue()?` $\succ$ `U` and then `<async>.queue()?.{drain,saturated}` $\succ$ `U.drain`. Since `<async>.queue().{drain,saturated}` $\not\succ$ `U.drain`, the match between p and E is classified as low confidence.

Example 19. Assume p is the pattern `call <lodash>.dropWhile [3,3] 2:function` from Example 17 and E is the call expression in this function:

```
function f(x) {
  ...
  return _.dropWhile(arr, pred, x)
  ...
}
```

Here, p matches E. The analysis can trivially determine that 3 arguments are present, but it cannot determine whether the last argument has type `function`, so this become a low confidence match.

Example 20. The breaking change in *commander* mentioned in Example 5 affects the client *uglify-js*, which contains this code:

```
var program = require('commander');
var UglifyJS = require('../tools/node');
...
options.parse = program.parse;
...
UglifyJS.parse(...)
```

Our simple field-based alias analysis cannot distinguish between the `parse` property from *commander* and the `parse` property from the *uglify-js* module. The set of access paths inferred for the expression `UglifyJS.parse` is therefore {`<../tools/node>.parse`, `<commander>.parse`}, which gives a low confidence match with the pattern `call <commander>**.parse`, resulting in a false positive.

7 EVALUATION

We have implemented TAPIR in TypeScript using *acorn*[14] and *recast*[15] for producing and traversing ASTs. Our implementation of the static analysis is less than 1 000 LoC, including the alias analysis, the access path inference, and the pattern matcher. We evaluate TAPIR by answering the following research questions:

RQ1: How many of the breaking changes mentioned in the changelogs of widely used npm packages can be expressed using our pattern language, and how complex are the patterns?

RQ2: What are the recall (high recall means few false negatives), the precision (high precision means few false positives), the analysis confidence compared to true and false positives, and the running time of the TAPIR static analysis when applied to real-world clients?

Benchmark selection. To answer the research questions, we base our experiments on the 10 major updates of library packages from the npm registry that we also considered in our preliminary study in Section 3. Furthermore, we include 5 additional major updates of other libraries to reduce bias towards the types of breaking changes observed in the preliminary study. The 15 libraries and a summary of the experimental results are shown in Tables 2–5.

To address RQ2, we extracted client packages from the npm registry that depend on one or more of the libraries in the major version prior to the one for which the patterns are written for. For example, for the *lodash* version 4.0.0 update, we collected all packages in the npm registry

[14] https://www.npmjs.com/package/acorn
[15] https://www.npmjs.com/package/recast

Table 2. Overview of detection patterns.

Library	#BC	#Patterns	#Import	#Read	#Write	#Call
lodash 4.0.0	51	113	17	17	0	79
async 3.0.0	4	6	1	1	1	3
express 4.0.0	18	21	0	10	1	10
chalk 2.0.0	3	3	0	3	0	0
bluebird 3.0.0	8	10	1	2	0	7
uuid 3.0.0	1	1	0	0	0	1
commander 3.0.0	3	3	0	0	0	3
rxjs 6.0.0	23	36	19	10	0	7
core-js 3.0.0	26	35	19	12	0	4
yargs 14.0.0	1	1	0	0	0	1
node-fetch 2.0.0	9	9	1	1	1	6
winston 3.0.0	20	22	0	12	0	10
redux 4.0.0	2	2	1	0	0	1
jsonwebtoken 8.0.0	4	4	0	0	0	4
mongoose 5.0.0	14	17	0	0	5	12
Total	**187**	**283**	**59**	**68**	**8**	**148**

that depend on any version of *lodash* between version 3.0.0 and 3.10.1 (3.10.1 is the last version before 4.0.0). We then retrieved the GitHub repository for each package and found the git commit matching the required version of the client (the newest version where the benchmark update has not been applied). We discarded a package if a GitHub repository was not available, we could not identify the right commit, the repository contained no test suite, or running the test suite did not succeed.

For measuring recall we are interested in client packages whose test suite fails after switching to the new version of the library (without making any changes to the client code). For each of the libraries, we attempted to extract at least 10 such clients (a few more for *lodash* since that update contains many breaking changes). For some libraries, we could not find that many clients, typically because the latest major update is so old that there are few packages that ever depended upon versions before it, or because the breaking changes are relatively benign and therefore unlikely to cause any test failures. As result, we obtained 115 such clients (Table 4). For measuring precision and performance, we collected 150 additional clients (Table 5) whose test suites are unaffected by switching to the new version of the library.

7.1 RQ1 (Expressiveness)

For each of the breaking changes in the 15 package updates, we created corresponding API access patterns (apart from one case, which we explain later). In some cases, multiple patterns were required to describe a single breaking change. Consider, for example, the breaking change concerning the removal of the `thisArg` argument from many *lodash* methods, which (among other patterns) requires both the patterns from Example 4 and Example 17.[16] This particular breaking change requires 9 patterns since the `thisArg` argument can either be the second, third, or fourth argument of a method, some methods are chainable, and some can be imported from different modules. However, the breaking change affects 64 different methods, so being able to express it with just 9 patterns is acceptable.

[16] Both examples are shortened versions of the actual patterns since more methods are affected, which is expressed with $.\{f_1, \ldots, f_n\}$ patterns.

Table 3. Detection pattern statistics.

		Feature usage (mean appearance per detection pattern)								
Library	**Length**	`<M>`	`,`	`\`	`()`	`**`	`.`	`?`	`[m, n]`	TypeFilter
lodash 4.0.0	3.56	1.53	0.50	0.00	0.08	0.06	0.56	0.00	0.82	0.32
async 3.0.0	3.83	1.67	0.67	0.00	0.17	0.00	1.00	0.17	0.33	0.00
express 4.0.0	3.95	1.00	0.00	0.00	0.05	0.71	1.00	0.76	0.40	0.50
chalk 2.0.0	2.00	1.00	0.00	0.00	0.00	0.00	1.00	0.00	-	-
bluebird 3.0.0	3.60	1.00	0.00	0.00	0.00	0.60	0.90	0.60	0.57	0.14
uuid 3.0.0	2.00	1.00	0.00	0.00	0.00	0.00	1.00	0.00	0.00	0.00
commander 3.0.0	3.67	1.00	0.00	0.00	0.00	0.67	1.00	0.00	0.33	0.67
rxjs 6.0.0	2.11	1.08	0.03	0.06	0.00	0.08	0.61	0.08	0.00	0.57
core-js 3.0.0	2.03	1.14	0.14	0.00	0.00	0.14	0.37	0.23	0.00	0.00
yargs 14.0.0	4.00	1.00	0.00	0.00	0.00	1.00	1.00	0.00	1.00	0.00
node-fetch 2.0.0	3.44	1.00	0.00	0.00	0.00	0.67	0.78	0.67	0.33	0.17
winston 3.0.0	3.73	1.00	0.00	0.00	0.05	0.36	1.36	0.36	0.90	0.40
redux 4.0.0	2.50	1.00	0.00	0.00	0.00	0.50	0.50	0.50	0.00	0.00
jsonwebtoken 8.0.0	3.00	1.00	0.00	0.00	0.00	0.00	1.00	0.00	1.00	0.00
mongoose 5.0.0	4.53	1.00	0.00	0.00	0.06	0.88	1.06	0.88	0.50	0.42
Total	**3.25**	**1.25**	**0.23**	**0.01**	**0.05**	**0.24**	**0.71**	**0.23**	**0.66**	**0.32**

When performing the experiments for RQ2, we discovered 5 breaking changes in *rxjs*, 2 in *express*, 2 in *winston*, 1 in *bluebird* and 1 in *node-fetch* that were not mentioned in the changelogs. We have also written patterns for those changes. The total number of breaking changes for each library is shown in the #**BC** column in Table 2.

In total, we have written 283 patterns, where 15 are for the 11 breaking changes we found and the remaining 268 are for the 176 breaking changes mentioned in the changelogs, so on average each breaking change amounts to 1.5 patterns. The number of patterns written for each benchmark varies heavily (see the #**Patterns** column in Table 2), from 1 for *uuid* to 113 for *lodash*, but this is well-aligned with the differences in the number of breaking changes per update as observed in Section 3. The columns #**Import**, #**Read**, #**Write**, and #**Call** show the number of patterns of the different kinds.

For one of the breaking changes in the update of *core-js* to version 3.0.0, we were not able to write a pattern capturing the affected API. This breaking change concerns the removal of iterators from the `Number` object, which allowed users to write code such as `for (var i of 3)` to iterate through the numbers 0 to 2. In principle, we could extend the language and the analysis to handle cases like this, but that would increase the complexity of writing patterns considerably, without providing much benefit to the user. We have not found any occurrences of this unexpressible breaking change while conducting the experiments.

As explained in Section 5, the patterns are generally short and simple. Table 3 shows the mean length of the patterns and the mean number of occurrences of each kind of pattern construct per pattern. More specifically, we count the number of *AccessPathPattern*, *PropertyPathPattern*, and *Filter* derivations in the parse tree of each pattern according to the grammar of Figure 3. For example, the path in Example 1 has length 1, the path in Example 3 has length 4, and the path in Example 4 has length 3. The mean length of each pattern is only 3.25, which shows that patterns are typically quite small.

With this length metric, we choose to count a property path pattern with multiple property names as one (and similarly for type filters containing multiple types, and for globs at import patterns), because those do not contribute much to the pattern complexity. For example, the `thisArg` breaking change of *lodash* affects 64 different methods, which leads to a property path pattern with

Table 4. Experimental results for clients with test suites that fail after the update.

Library	#Clients	Recall	TP	$\text{TP}_{\neg B}$	TP_B	FP	Precision	High conf.
lodash 4.0.0	14	100%	83	74	9	2	98%	78
async 3.0.0	10	100%	10	10	0	2	83%	10
express 4.0.0	10	100%	60	60	0	8	88%	30
chalk 2.0.0	10	100%	54	54	0	0	100%	54
bluebird 3.0.0	10	100%	33	8	25	0	100%	24
uuid 3.0.0	1	100%	2	2	0	0	100%	2
commander 3.0.0	10	100%	23	22	1	12	66%	21
rxjs 6.0.0	10	100%	464	464	0	73	86%	294
core-js 3.0.0	10	100%	23	23	0	0	100%	22
yargs 14.0.0	1	100%	1	1	0	0	100%	1
node-fetch 2.0.0	8	100%	51	9	42	35	59%	6
winston 3.0.0	10	100%	59	50	9	10	86%	28
redux 4.0.0	4	100%	44	3	41	4	92%	7
jsonwebtoken 8.0.0	2	100%	6	0	6	0	100%	6
mongoose 5.0.0	5	100%	41	1	40	4	91%	10
Total	**115**	**100%**	**954**	**781**	**173**	**150**	**86%**	**593**

64 property names, but finding that list of names was trivial, and the pattern did not take more time to write than other patterns.

The majority of the patterns use only a few features of the pattern language. Not surprisingly, import path patterns (`<M>`), property path patterns (`.`), and argument count filters (`[m, n]`) are used frequently. Type filters (TypeFilter), property chains (`**`), and disjunctions (`,`) are also used for some libraries but less often. The exclusion operator (\) and function return patterns (`()`) are used only in very few cases.

As discussed in Section 6, deciding whether to use p or $p?$ depends on the typical usage pattern of the API. In cases where an access path pattern identifies an API on an object where that object serves as a form of namespace, we avoid using the latter since TAPIR can construct access paths for these usages with high precision. For more dynamic cases where objects are constructed and commonly passed around in the client code, we use the more conservative variant $p?$. For all the constructed patterns, we found it easy to determine which of these categories the API usage pertained to. We ended up using the $p?$ variant in 61 of the 283 patterns.

The patterns shown in Examples 1–20 are representative of those we have used in the evaluation. For the full list of patterns (and corresponding changelog descriptions), see https://brics.dk/tapir/.

For future work, it may be interesting to perform a user study, involving for example the library developers in the process of writing detection patterns for breaking changes. It is of course to our advantage that, having designed TAPIR, we are experts in the pattern language; on the other hand, we have no prior knowledge of the library code and the changes made in the updates. In our experience, the difficulty of writing patterns lies in comprehending the changelogs. Once we understood exactly which functions/properties/modules a specific breaking change concerned, and under which conditions it was relevant, writing the pattern usually took only a few seconds. We believe that this task will be much easier for the package maintainers since they already have the domain knowledge. As an added benefit, writing the patterns forces the package developers to be more explicit about the scope of breaking changes. For example, the *lodash* package maintainers would have to explicitly state which methods are affected by the `thisArg` removal breaking change, which would aid client developers in the update process.

In conclusion, the API access point pattern language can express almost every kind of breaking change observed in practice, and the patterns are generally quite simple.

Table 5. Experimental results for clients with test suites that are unaffected by the update.

Library	#Clients	TP	TP$_{\neg B}$	TP$_B$	FP	Precision	High conf.
lodash 4.0.0	10	5	0	5	0	100%	5
async 3.0.0	10	0	0	0	0	100%	0
express 4.0.0	10	51	51	0	5	91%	30
chalk 2.0.0	10	1	1	0	0	100%	1
bluebird 3.0.0	10	47	18	29	0	100%	42
uuid 3.0.0	10	0	0	0	0	100%	0
commander 3.0.0	10	16	16	0	0	100%	16
rxjs 6.0.0	10	112	111	1	17	87%	52
core-js 3.0.0	10	30	30	0	0	100%	30
yargs 14.0.0	10	3	3	0	0	100%	3
node-fetch 2.0.0	10	48	0	48	8	86%	19
winston 3.0.0	10	24	19	5	37	39%	9
redux 4.0.0	10	31	0	31	1	97%	15
jsonwebtoken 8.0.0	10	64	5	59	0	100%	61
mongoose 5.0.0	10	39	0	39	6	87%	25
Total	**150**	**471**	**254**	**217**	**74**	**86%**	**308**

7.2 RQ2 (Recall, Precision, Confidence, and Performance)

Recall. For tapir to be useful in practice, the recall must be high, as discussed in Section 1: it is important that it does not miss places in the client code that require changes when updating to the new version of the library. To measure the recall, we need a collection of clients that are known to be affected by breaking changes. For this purpose we use the 115 client packages whose test suites fail when switching to the new version of the library without adapting the client code.

We manually investigated the cause of each of the failing tests, and then modified the client code to adapt to the new version of the library. We then reran the tests and confirmed that the tests succeeded. Finally, we compared the set of all the source code locations that we had to modify in this process with the set of source locations reported by tapir. As result, all the source code locations that we had to manually modify were found by tapir, indicated by the 100% recall in Table 4.

Since the test suites do not have perfect coverage, this is of course not a guarantee that there are no false negatives. However, the fact that not a single test failure remains in the 115 client packages after fixing the source code locations suggested by tapir is a good indication that false negatives are uncommon. If running tapir on, for example, obfuscated code that uses dynamic property operations or `eval`, then it will perform poorly, but this experiment gives some confidence that recall is excellent for real-world JavaScript source code.

Precision. To measure the precision, we ran tapir on all 265 clients (without the code changes made for the recall experiment). For each alarm that did not match any of the manually patched source locations from the recall experiment, we manually investigated the alarm and recorded whether it was a true positive (i.e., the source location actually involves the API access point) or a false positive (shown as **TP** and **FP**, respectively, in Tables 4 and 5).

For some of the breaking changes (primarily belonging to the behavioral category from Table 1), tapir cannot decide with certainty whether a call is affected by the breaking change, either because the pattern language is too coarse-grained, or because the client application is itself a library, which makes it impossible to determine if some client uses that library in a way that involves the breaking change. We conservatively mark alarms of this form as true positives as long as tapir points to the correct function and it satisfies the filters, but we report them separately as $\mathbf{TP}_B$ in Tables 4 and 5 since we acknowledge that they may not actually be true positives in all cases.

As shown in Table 4, for the group of clients whose test suites detect breaking changes in the libraries, 86% of all alarms are true positives (**Precision**), showing that the precision of TAPIR is high in practice. Only a few of the alarms (18%) belong to the more uncertain $\mathbf{TP}_B$ category, which means that the remaining 82% of alarms ($\mathbf{TP}_{\neg B}$) point to places in the client source code that with certainty have to be patched to update the benchmark.

Unsurprisingly, the number of alarms reported per client is significantly higher for the clients whose test suites are affected by the library update than the other group of clients (9.6 per client compared to 3.6 per client).

For 76 of the 150 clients in the second group, TAPIR reports no alarms at all. For example, no alarms are reported for any of the 10 *async* clients. This demonstrates an important point: even though most breaking changes in libraries affect *some* clients, each individual client is unlikely to be affected by a specific breaking change.

The average precision of TAPIR for the second group of clients is 86%. For *winston* the precision is only 39%, but that is mostly due to one breaking change affecting the `info`, `warn`, and `error` methods. This breaking change only occurs when the second argument is a function, but for most calls to these methods, TAPIR is unable to determine the type of that argument and therefore reports a low confidence alarm. For 9 of the 15 libraries, the precision is 100%.

The false positives that TAPIR reports are mostly cases where the relaxed pattern variant p? is used together with a breaking change that affects a relatively common property name such as `map` or `catch`. Using the strict pattern p instead would reduce the recall, and the false positives are easily identified as false positives, so using the relaxed form is the best trade-off. The remaining false positives are due to imprecise checking of `call` filters or objects being mixed together due to imprecision in the field-based analysis.

In conclusion, TAPIR reports only a modest number of false positives, for both groups of clients and both groups of libraries.

Confidence. 54% of the alarms in the affected test suite clients and 57% of the alarms in the unaffected test suite clients are marked as high confidence by the analysis mechanism described in Section 6.4. We have not seen any high confidence alarms in the false positive category, which shows that for around half of the reported alarms, in practice the user does not even have to consider the possibility of false positives. Notice that low confidence alarms do not coincide with the uncertainty alarms ($\mathbf{TP}_B$). A low confidence alarm occurs because the analysis is too imprecise to either prove or disprove that a source location is affected by a breaking change. An uncertain alarm occurs whenever TAPIR lacks information on how the potentially affected API is used, or the detection pattern language is too coarse-grained to capture the constraints for the breaking change.

In conclusion, the analysis confidence reporting mechanism cuts the number of alarms that must be considered as potential false positives in half, thereby reducing the manual overhead of using TAPIR considerably. As discussed above, the false positive rate is already low, and with this mechanism it effectively becomes even lower.

Performance. Running TAPIR on the source code of the 265 clients takes 4 minutes and 20 seconds in total; in other words, the static analysis is clearly fast enough for practical use.

8 RELATED WORK

Software evolution studies. Several papers have investigated the evolution of software libraries, for example how and why breaking changes are introduced in library updates [Brito et al. 2018; Dig and Johnson 2006; Koçi et al. 2019; Mezzetti et al. 2018; Mitropoulos et al. 2019]. Dig and Johnson

[2006] find that more than 80% of all breaking changes in Java seem like refactorings from the library's perspective (renames, method split-ups, moves, etc.). This is similar to our findings for JavaScript from Section 3.

Multiple studies have investigated why clients do not keep their dependencies up-to-date [Derr et al. 2017; Zerouali et al. 2019]. Concerns about breaking changes are for the majority of developers a strong reason for not updating dependencies, which motivates the creation of tools like TAPIR.

Breaking change detection and patching tools. Automatically detecting locations in client code affected by breaking changes has been pursued by previous work [Chow and Notkin 1996; Dagenais and Robillard 2011; Fazzini et al. 2019; Kang et al. 2019; Li et al. 2018; Nguyen et al. 2010; Padioleau et al. 2006; Zhang et al. 2020]. Several techniques also include mechanisms for patching the affected client code [Chow and Notkin 1996; Dagenais and Robillard 2011; Kang et al. 2019; Nguyen et al. 2010; Padioleau et al. 2006] and even how to automatically infer broken APIs from the diff between library versions [Dagenais and Robillard 2011; Fazzini et al. 2019; Li et al. 2018; Nguyen et al. 2010]. Common amongst these papers is that they all target statically typed languages such as C or Java, which makes identifying API usages and extracting type information much simpler. To the best of our knowledge, the only exception is the PyCompat tool for Python [Zhang et al. 2020]. While Python is also dynamically typed, it is still significantly different from JavaScript. The API usage in Python is sufficiently static for PyCompat to be able to identify the relevant call sites in the client code directly from the AST. For JavaScript, a more sophisticated mechanism, such as the access path analysis of TAPIR, is required to achieve good precision.

Lightweight static analysis. Creating sound context-sensitive dataflow analyses for JavaScript is notoriously difficult due to the highly dynamic language features [Kristensen and Møller 2019; Stein et al. 2019]. For that reason, using lightweight, unsound analysis techniques has been explored by previous work, for example to help with refactoring [Feldthaus and Møller 2013] and for constructing call graphs [Feldthaus et al. 2013]. Common for these approaches is that they are unsound but work well in practice. As shown in Section 7, the static analysis of TAPIR belongs to this family of analyses.

A disadvantage of our pattern language is that pattern writers must consider whether to use the relaxed variant of patterns (i.e., p?). As we argue in Section 7, this choice is usually simple, but ideally the pattern writer should not have to make it. For future work, we plan to explore whether a more powerful analysis can remove the need for the relaxed variant, however, it is not easy to accomplish that while preserving the essential properties of our current approach: (1) it avoids false negatives, (2) it has an acceptable number of false positives, and (3) it is fast. In particular, using the analysis technique of Feldthaus et al. [Feldthaus et al. 2013] would not allow us to drop the relaxed variant of patterns. Although using that technique could perhaps improve precision, it is too unsound, meaning that we would lose the excellent recall of our current approach.

9 CONCLUSION

Many npm packages depend on heavily outdated dependencies, which undermines security and prevents bug fixes and other improvements from reaching the end users. We have presented a simple pattern language that allows library developers to easily express which parts of the API are affected by breaking changes in major updates. To help clients adapt their code to the breaking changes, we have developed the tool TAPIR that finds the relevant locations in the client code.

Our evaluation shows that the pattern language has sufficient expressiveness to cover the breaking changes described in changelogs. Only 283 patterns are required to identify 187 breaking changes affecting 15 top npm libraries. By using these patterns, TAPIR successfully locates the instances of the patterns in 265 clients, with only one in seven alarms being false positives, and zero false negatives. TAPIR can mark around half of the reported alarms as high confidence, and those are likely true positives, which reduces the manual overhead even further. Furthermore, it takes TAPIR on average less than a second to analyze the source code of a client. As result, TAPIR can relieve the client developers from the often difficult and time-consuming task of comprehending the changelogs and finding the affected locations in their code when updating dependencies.

Once the affected locations have been found in the client code, the next task for the developer is to update those parts of the code (if any), to adapt to the breaking changes in the library. That task can possibly also be partly automated, which we will explore in future work. We also plan to investigate how to automatically generate API access point patterns for breaking changes, possibly using NoRegrets+ [Møller and Torp 2019] as a starting point.

ACKNOWLEDGMENTS

This work was supported by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No 647544).

REFERENCES

Aline Brito, Laerte Xavier, André C. Hora, and Marco Tulio Valente. 2018. Why and how Java developers break APIs. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER 2018, Campobasso, Italy, March 20-23, 2018*. IEEE Computer Society, 255–265.

Kingsum Chow and David Notkin. 1996. Semi-automatic update of applications in response to library changes. In *1996 International Conference on Software Maintenance (ICSM '96), 4-8 November 1996, Monterey, CA, USA, Proceedings*. IEEE Computer Society, 359.

Barthélémy Dagenais and Martin P. Robillard. 2011. Recommending Adaptive Changes for Framework Evolution. *ACM Trans. Softw. Eng. Methodol.* 20, 4 (2011), 19:1–19:35.

Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. 2017. Keep me Updated: An Empirical Study of Third-Party Library Updatability on Android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*. ACM, 2187–2200.

Danny Dig and Ralph E. Johnson. 2006. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance* 18, 2 (2006), 83–107.

Mattia Fazzini, Qi Xin, and Alessandro Orso. 2019. Automated API-usage update for Android apps. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2019, Beijing, China, July 15-19, 2019*. ACM, 204–215.

Asger Feldthaus and Anders Møller. 2013. Semi-automatic rename refactoring for JavaScript. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2013, part of SPLASH 2013, Indianapolis, IN, USA, October 26-31, 2013*. ACM, 323–338.

Asger Feldthaus, Max Schäfer, Manu Sridharan, Julian Dolby, and Frank Tip. 2013. Efficient construction of approximate call graphs for JavaScript IDE services. In *35th International Conference on Software Engineering, ICSE '13, San Francisco, CA, USA, May 18-26, 2013*. IEEE Computer Society, 752–761.

Hong Jin Kang, Ferdian Thung, Julia Lawall, Gilles Muller, Lingxiao Jiang, and David Lo. 2019. Semantic Patches for Java Program Transformation (Experience Report). In *33rd European Conference on Object-Oriented Programming, ECOOP 2019, July 15-19, 2019, London, United Kingdom. (LIPIcs)*, Vol. 134. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 22:1–22:27.

Rediana Koçi, Xavier Franch, Petar Jovanovic, and Alberto Abelló. 2019. Classification of Changes in API Evolution. In *23rd IEEE International Enterprise Distributed Object Computing Conference, EDOC 2019, Paris, France, October 28-31, 2019*. IEEE, 243–249.

Erik Krogh Kristensen and Anders Møller. 2019. Reasonably-most-general clients for JavaScript library analysis. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. IEEE / ACM, 83–93.

Li Li, Tegawendé F. Bissyandé, Haoyu Wang, and Jacques Klein. 2018. CiD: automating the detection of API-related compatibility issues in Android apps. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, Amsterdam, The Netherlands, July 16-21, 2018*. ACM, 153–163.

Gianluca Mezzetti, Anders Møller, and Martin Toldam Torp. 2018. Type Regression Testing to Detect Breaking Changes in Node.js Libraries. In *32nd European Conference on Object-Oriented Programming, ECOOP 2018, July 16-21, 2018, Amsterdam, The Netherlands (LIPIcs)*, Vol. 109. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 7:1–7:24.

Dimitris Mitropoulos, Panos Louridas, Vitalis Salis, and Diomidis Spinellis. 2019. Time present and time past: analyzing the evolution of JavaScript code in the wild. In *Proceedings of the 16th International Conference on Mining Software Repositories, MSR 2019, 26-27 May 2019, Montreal, Canada*. IEEE / ACM, 126–137.

Anders Møller and Martin Toldam Torp. 2019. Model-based testing of breaking changes in Node.js libraries. In *Proceedings of the ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT FSE 2019, Tallinn, Estonia, August 26-30, 2019*. ACM, 409–419.

Hoan Anh Nguyen, Tung Thanh Nguyen, Gary Wilson Jr., Anh Tuan Nguyen, Miryung Kim, and Tien N. Nguyen. 2010. A graph-based approach to API usage adaptation. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2010, October 17-21, 2010, Reno/Tahoe, Nevada, USA*. ACM, 302–321.

Yoann Padioleau, René Rydhof Hansen, Julia L. Lawall, and Gilles Muller. 2006. Semantic patches for documenting and automating collateral evolutions in Linux device drivers. In *Proceedings of the 3rd Workshop on Programming Languages and Operating Systems: Linguistic Support for Modern Operating Systems, PLOS 2006, San Jose, California, USA, October 22, 2006*. ACM, 10.

Yoann Padioleau, Julia L. Lawall, René Rydhof Hansen, and Gilles Muller. 2008. Documenting and automating collateral evolutions in linux device drivers. In *Proceedings of the 2008 EuroSys Conference, Glasgow, Scotland, UK, April 1-4, 2008*. ACM, 247–260.

Benno Stein, Benjamin Barslev Nielsen, Bor-Yuh Evan Chang, and Anders Møller. 2019. Static analysis with demand-driven value refinement. *PACMPL* 3, OOPSLA (2019), 140:1–140:29.

Ahmed Zerouali, Tom Mens, Jesús M. González-Barahona, Alexandre Decan, Eleni Constantinou, and Gregorio Robles. 2019. A formal framework for measuring technical lag in component repositories - and its application to npm. *Journal of Software: Evolution and Process* 31, 8 (2019).

Zhaoxu Zhang, Hengcheng Zhu, Ming Wen, Yida Tao, Yepang Liu, and Yingfei Xiong. 2020. How Do Python Framework APIs Evolve? An Exploratory Study. In *27th IEEE International Conference on Software Analysis, Evolution and Reengineering, SANER London, Ontario, February 18-21, 2020*. IEEE, 81–92.

Markus Zimmermann, Cristian-Alexandru Staicu, Cam Tenny, and Michael Pradel. 2019. Small World with High Risks: A Study of Security Threats in the npm Ecosystem. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*. USENIX Association, 995–1010.