# Type Regression Testing to Detect Breaking Changes in Node.js Libraries

**Gianluca Mezzetti**
Aarhus University, Denmark
mezzetti@gmail.com

**Anders Møller**
Aarhus University, Denmark
amoeller@cs.au.dk

**Martin Toldam Torp**
Aarhus University, Denmark
torp@cs.au.dk

## Abstract

The npm repository contains JavaScript libraries that are used by millions of software developers. Its semantic versioning system relies on the ability to distinguish between breaking and non-breaking changes when libraries are updated. However, the dynamic nature of JavaScript often causes unintended breaking changes to be detected too late, which undermines the robustness of the applications.

We present a novel technique, *type regression testing*, to automatically determine whether an update of a library implementation affects the types of its public interface, according to how the library is being used by other npm packages. By leveraging available test suites of clients, type regression testing uses a dynamic analysis to learn models of the library interface. Comparing the models before and after an update effectively amplifies the existing tests by revealing changes that may affect the clients.

Experimental results on 12 widely used libraries show that the technique can identify type-related breaking changes with high accuracy. It fully automatically classifies at least 90% of the updates correctly as either major or as minor or patch, and it detects 26 breaking changes among the minor and patch updates.

**2012 ACM Subject Classification** Software and its engineering → Software libraries and repositories

**Keywords and phrases** JavaScript, semantic versioning, dynamic analysis

**Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2018.7

**Supplement Material** ECOOP Artifact Evaluation approved artifact available at
`http://dx.doi.org/10.4230/DARTS.4.3.8`

**Funding** 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).

## 1 Introduction

The world's largest software repository, npm,[1] hosts 475 000 Node.js JavaScript packages as of January 2018 and is used by millions of software developers. Most packages are libraries,

[1] `https://www.npmjs.com/`

32nd European Conference on Object-Oriented Programming (ECOOP 2018).
Editor: Todd Millstein; Article No. 7; pp. 7:1–7:24
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

and many are frequently updated, so versioning is essential to ensure that the components of a software system are compatible and up-to-date. The npm system encourages the use of *semantic versioning* [25], which distinguishes between patch, minor and major version updates: patch and minor are incremental updates that are not supposed to break any library client, whereas major version updates do not have any restrictions.

Unfortunately, the distinction between breaking and non-breaking changes is not always clear, and it may be difficult for the library developer to decide how to increment version numbers when changes are released. The dynamic nature of the JavaScript ecosystem often causes library developers to erroneously believe that their updates cannot break any clients. A prominent example was the supposedly minor update of a package called *debug* from version 2.3.3 to 2.4.0 on December 14 2016. Due to a simple spelling error, all clients that tried to load the new version crashed immediately.[2] The bug was fixed within an hour, but *debug* was downloaded more than 27 million times in December 2016 alone, which means that it may still have affected thousands of installations.

To make matters worse, Node.js library interfaces are rarely specified precisely, so library developers and their clients may have different views on which aspects of the library are supposed to be internal to the library and which aspects the client code may rely on. As an example, the developers of the popular package *React* from version 15.3.2 to 15.4.0 reorganized the module react/lib/ReactMount that was intended for internal use only, but numerous other packages used that module and therefore broke.[3]

In statically typed programming languages like Java, even without versioning systems, the type system is helpful for detecting many situations where an upgrade of a library causes an application to break. For example, if the type signature of a library method has changed, the application code no longer compiles. Using access modifiers enables library developers to encapsulate private parts of the library, so that internal data representations and operations can be changed without affecting client code. Additionally, annotations about deprecated functionality signal to the application developer that attention is needed. Java's binary compatibility conditions [13, Chapter 13] and tools like JAPICC [23] make it possible to detect type-related breaking changes in libraries even without involving the client code. In the world of JavaScript and npm, there is no static type system or compilation to binary code, so incompatibility issues are often not detected until runtime.

We distinguish between two main categories of breaking changes. *Type-related* breaking changes are modifications of a library that affect the presence or types of functions or other properties in the library interface. Such changes include renaming a public function, moving it to another location, or changing its type signature. Type-related breaking changes should evidently always be reflected as major version updates. As the library interface is partly defined by the library initialization code, initialization errors like the one in the *debug* example are generally categorized as type-related breaking changes. *Semantic* breaking changes are modifications that are not type-related but affect the library functionality in other ways that may cause clients to malfunction. This category is more blurry, as it depends on a semantic contract between the library and the clients. The type checker in Java detects type-related breaking changes, but not semantic ones. Our goal is to provide a mechanism that can similarly detect type-related breaking changes for Node.js JavaScript libraries, without requiring type annotations.

[2] `https://github.com/visionmedia/debug/issues/347`

[3] `https://github.com/supnate/rekit/issues/16`

In this paper, we first present a preliminary study of real-world breaking changes in the npm repository. The study shows that breaking changes do occur at patch and minor updates, and that a significant portion of the breakage is type-related. Next, we propose a technique, called *type regression testing*, to automatically detect such type-related breaking changes, which we call *type regressions*, thereby gaining some of the benefits that are known from statically typed languages.

Our type regression testing technique is based on a novel dynamic type analysis that automatically learns relevant information about the API of a given library. The basic idea is quite simple: The npm repository makes it possible to identify packages that directly or transitively depend on the library of interest. (For example, the *lodash* library has more than 50 000 direct dependents.) By exercising the test suites for those packages, we can monitor the dynamic execution and construct a model of the library API. When the library implementation has been updated and a new version is about to be released, the test suites are run again, this time with the updated library to infer a new model of the new API. We then compare the new and the old API models using certain rules to identify breaking changes in the update. Importantly, we do not use the library's own test suite, but the test suites of the clients, because they are more likely to provide representative executions and only use the public parts of its API. Type regression testing *amplifies* the existing test suites: even if the tests do not fail, it can identify type-related changes in the interactions between the clients and the library.

Making this idea work in practice requires a suitable notion of models of library APIs, together with a mechanism for comparing models before and after the library implementations have been updated. The API modeling and the comparison mechanism need to be aligned with how JavaScript library developers usually organize their code and try to adhere to the semantic versioning guidelines.

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

- We first present a preliminary experimental study on the prevalence and the kinds of breaking changes in the npm repository. We find that at least 5% of all packages have been affected by a breaking change in a minor or patch update of a dependency, and that a majority of these breaking changes are due to changes in the public API of the package.
- We propose *type regression testing* as a mechanism for leveraging the preexisting test suites of npm packages that depend on a JavaScript library of interest, to learn models of the library API and detect likely breaking changes.
- At the core of the type regression testing mechanism, library APIs are modeled using *dynamic access paths* and types that provide information about how the library and the clients interact. We define precisely how these models are obtained and compared. The possible breaking changes are identified by *type regressions*: changes in the type signatures of the library APIs that are incompatible with the mutual expectations of client and library developers.
- We report results from an experimental evaluation of the approach on 12 of the most depended upon libraries in the npm system, demonstrating that it can detect type-related breaking changes with high accuracy. Our implementation, named NoRegrets, classifies at least 90% of the updates correctly as either major or as minor or patch, and it also successfully identifies 26 breaking changes among the minor and patch updates. Moreover, in cases where a likely breaking change is detected, the warning message produced by the tool pinpoints the involved part of the library, which aids diagnosis.

## 2 A Preliminary Experimental Study of Breaking Changes

To understand how frequently library updates break client applications, and what the typical causes of the breaking changes are, we conducted a large experiment on npm package updates. For this experiment, we exploited the fact that many clients have test suites and that if a test fails after updating a dependency, then the failure indicates that the update contains a breaking change.

To serve as clients, a sample consisting of 4616 packages with test suites was randomly picked from npm. Most of these packages are intended to be used as libraries, but they still depend upon other libraries and have test suites as required for this study.

All npm packages include a configuration file called package.json where the package dependencies are specified. Each dependency specification consists of a package name and a versioning constraint. This constraint specifies the range of versions which the dependency must lie within. The npm system will always install the newest version of the dependency satisfying this versioning constraint. According to the semantic versioning guidelines, clients should use constraints that permit all minor and patch updates but no major updates. Thereby the client will automatically receive the bug fixes and other improvements introduced in minor and patch updates, but hopefully never break since breaking changes are only allowed in major updates.

For each dependency specified in the package.json file, we ran the test suites using each version of the dependency going from the oldest to the newest version satisfying the semantic versioning constraint. We removed versions which, although satisfying the semantic versioning constraint, would never have been installed by npm in practice since a newer version was already available at the point where the constraint was created. An update was flagged as potentially breaking whenever the test suite of the client went from all tests passing to at least one test failing after applying the update. We did not count at the granularity of individual test cases, since it is technically difficult to do because of nested and parameterized tests.

This amounts to 75 913 executions of test suites of the 4616 packages. Of these, 430 fail, meaning that breaking changes are detected. Whenever we had two versions of the same client appearing among the failures, we discarded the oldest one of them to avoid duplicates. We also discarded major updates and updates containing a pre-release identifier. A pre-release identifier is a hyphen followed by a tag added to the end of a version number used to indicate, for example, release candidates and beta versions. Updates containing a pre-release identifier should be treated as a major updates according to the semantic versioning principle. Furthermore, we chose to exclude 94 failures that could not be reproduced consistently due to flaky tests in the clients. With these filters, the total number of failures was reduced to 263 affecting 259 different clients. None of these 263 breaking changes should have appeared if semantic version had worked as intended and the library developers and the client developers had a common agreement on what is the public interface of the libraries!

Thus, at least 5% of the npm packages have experienced a breaking change due to a non-major dependency update. The actual number is likely much higher, because not many packages have test suites that are thorough enough to catch all breaking changes in libraries they depend on.

Next, we manually categorized the test failures as either *type-related*, *semantic*, or *unknown*, the latter for the cases where we could not determine the cause within 30 minutes. We consider any update that modifies the presence or types of modules, properties, function arguments, and function return values as type-related. Specifically, an update that relocates a module to a different path typically causes attempts to load the module using the old path to fail. Any test failure that is not type-related is considered semantic. Thus, the type-related test failures roughly correspond to the kinds of errors that could be caught statically if using a language like Java with type checking instead of JavaScript.

As result we found that among the 263 failures, 176 were type-related, 37 were semantic, and the remaining 50 were marked as unknown. Some type-related breaking changes are easy to detect. In particular, sometimes simply attempting to load a library module fails, because it has been relocated or because its initialization code consistently crashes after an update. An example of the latter is the bug in the *debug* package mentioned in Section 1, which alone accounts for 101 of the failures. Even if we only count the occurrence of this bug once, we still find that at least 46% of the breaking changes are type-related.

This preliminary study motivates the need for tool support to detect breaking changes in Node.js libraries before the developers publish new versions of their libraries. It also justifies focusing on breaking changes that are type-related, which are more amenable to automated detection than the semantic ones.

To our knowledge the only similar tool is *dont-break*,[4] which follows essentially the approach we have used in our preliminary study: it detects breaking changes simply by running the test suites of library clients each time a new version of the library is about to be released. Although this is a simple approach, it has an important limitation: The library developer presumably has no knowledge of the clients, let alone their tests, so it can be difficult for him or her to identify the relevant parts of the library whenever a client test fails. In contrast, type regression testing precisely pinpoints the involved changes made at the library interface, allowing the library developer to decide whether or not the breaking change is intended, without having any knowledge of the client code. Additionally, type regression testing can detect breaking changes that do not necessarily surface as failing client tests and can therefore also be viewed as a test amplification mechanism.

## 3 Motivating Example

Consider the following subtle change of the `isIterateeCall` method in the *lodash* library when upgraded from 3.2.0 to 3.3.0—a minor update that should not introduce breaking changes.

```
// lodash 3.2.0
function isIterateeCall(value, index, object) {
  ....
  return prereq && object[index] === value;
}
```

```
// lodash 3.3.0
function isIterateeCall(value, index, object) {
  ....
  var other = object[index];
  return prereq && (value === value ? value === other : other !== other);
}
```

The variable `prereq` is computed in the same way in both versions. The important difference is that the `object[index]` property lookup is only executed when `prereq` is `true` in version 3.2.0, due to short-circuiting of `&&`, whereas it is always executed in version 3.3.0. If `index` is an object, then the `toString` method is implicitly called on `index` to coerce it to a string such

[4] https://www.npmjs.com/package/dont-break

that it can be used in the property lookup. However, it is possible that there is no `toString` method on `index` if, for example, the `index` value was created using `Object.create(null)`. Consequently, a type error exception will be thrown when the coercion is attempted.

The `isIterateeCall` method is not directly visible to the client code, but it is called internally by the public `merge` method that forwards one of its arguments as the `index` parameter of `isIterateeCall`. The `merge` method takes a target object and a variadic number of source objects and merges the properties of the source objects into the target object. The artificial library client in lines 13–18 witnesses the problem.

```
12 // client
13 var l = require('lodash');
14 var o1 = {};
15 var o2 = {};
16 var oBad = Object.create(null);
17 var o4 = {};
18 l.merge(o1, o2, oBad, o4); // Type error in lodash 3.3.0
```

The `oBad` object is passed as the `index` parameter to `isIterateeCall`, and a runtime type error appears when using *lodash* 3.3.0 but not when using the older version. The problem has been confirmed by the developers who fixed it in *lodash* 3.3.1 as mentioned in the changelog.[5]

Type regression testing automatically finds this problem as follows. First, it builds a model of the library API by observing the executions of the tests of clients, for both the old and the new library version. Second, it compares the two models to detect breaking changes. This particular breaking change is detected by observing that the *lodash* 3.3.0 model, unlike the one generated by version 3.2.0, requires the `toString` method to be present on the third argument of `merge`. This expectation is breaking since the third parameter is in a contravariant position and its type is more specific than before the update.

It is important to notice that type regression testing leverages and amplifies preexisting test suites. For this *lodash* bug, the breaking change is detected even though the execution of `merge` does not trigger the type error in any of the client tests.

Furthermore, even if one of the client tests had triggered the type error, that would only produce a stack trace indicating a problem in `isIterateeCall`, and manual effort would then be required to be connect this type error to the third argument of `merge`. In contrast, type regression testing identifies exactly the `toString` property on the third argument of `merge` as the source of the problem.

## 4 Overview

Type regression testing targets a specific use case: a library developer is ready to release a new library version and wants to know whether the new implementation introduces breaking changes. The workflow of type regression testing comprises two phases.

**(1) Public API Discovery.** In the first phase, all the packages with tests that depend on the old library version are retrieved from npm. The type regression in the motivating example can be spotted by using any client whose tests cause the invocation of the *lodash* `merge` function, for example *strong-params* at version 0.7.0.[6] Then, a public API model of both the old and the new library version is built using an instrumented interpreter that runs the

[5] https://github.com/lodash/lodash/wiki/Changelog#v331

[6] https://github.com/ssowonny/strong-params

tests of all the collected dependents. A model $\pi$, which we formally define in Section 5, is a map from *dynamic access paths* to *types*.

Intuitively, a dynamic access path is an abstraction of the sequence of actions that are performed to obtain a value. Types are an abstraction over values that is more specific than the ordinary notion of runtime types in JavaScript. For example, the path $p$ = `require`(*lodash*)`.merge(3)`$_4$`.toString` exhibits the regression in the example from Section 3. This path denotes any value obtained by accessing the `toString` property of the third argument passed to the `merge` function when `merge` is invoked with four arguments. No type is associated with this path in the execution of the tests of *strong-params* when using *lodash* 3.2.0, but a function is observed when switching to *lodash* 3.3.0. In the model $\pi$ of the public API of *lodash* 3.2.0, we have $\pi(p) = \circ$, denoting the fact that no value has been observed for $p$. In the model $\pi'$ of the public API of *lodash* 3.3.0, we instead have $\pi'(p) = \texttt{function} \wedge \texttt{object}$, which means that the value is an instance of `function` and `object` (formally $\pi'(p)$ is an intersection type). This phase is fully detailed in Section 5.

**(2) Type Regression Detection.** In the second phase, we compare the models $\pi$ and $\pi'$ obtained from the old and the new library version to report *type regressions*, which are indications of type-related breaking changes. Type regressions are detected by comparing $\pi(p)$ and $\pi'(p)$ for every dynamic access path $p$, using a notion of subtyping, which we define in Section 5.2. In the example above, the type regression is reported because the type $\texttt{function} \wedge \texttt{object}$ is not a supertype of $\circ$. This phase is fully detailed in Section 6.

# 5 Public API Discovery

The workflow described in Section 3 requires two models of the public API to be built, one for the old and one for the new library version. It is important that the API models only capture the publicly available API, so that the comparison is not susceptible to changes in the private parts of the library where modifications are always allowed.

In statically typed languages, like Java, the package structure, class hierarchy, and access modifiers statically identify the public API of a library. In a dynamically typed language, such as JavaScript, a library module is initialized by the execution of the library module itself. Specifically, in Node.js, the library code for initialization of a library *foo* is executed the first time `require("foo")` is called. The object stored by the library code in the `module.exports` variable is the one returned by the call to `require`. However, this object only exposes the immediately accessible part of the public API, and the public API often contains much more functionality. For example, all methods of router objects in the *express* library only become accessible after invoking `Router()`.[7] As this example illustrates, it is generally difficult to statically identify the public API, which is why we resort to dynamic analysis.

The model of the public API of a library is a map $\pi : \textbf{Path} \to \textbf{Type}$ assigning a type $t \in \textbf{Type}$ to each dynamic access path $p \in \textbf{Path}$. We now define dynamic access paths and types, and then we describe the discovery mechanism that builds $\pi$.

## 5.1 Dynamic Access Paths

Dynamic access paths, hereafter also shortened to paths, are used in the model to refer to values that are part of the interface between the client and the library. This mechanism

[7] `http://expressjs.com`

ignores the syntactical structure of the library code and only considers how the library is being accessed dynamically by the client code.

▶ **Definition 1** (Dynamic Access Path). A *dynamic access path* $p \in \textbf{Path}$ is a possibly empty sequence of *actions* $\alpha$ that abstractly represents a set of values, as defined by the grammar below. We indicate integers by the letters $i, j$ and strings (library names and property names) by the letter $n$.

$$p ::= \varepsilon \mid \texttt{require}(n) \mid p\alpha \qquad\qquad p \in \textbf{Path}$$

$$\alpha ::= .n \mid ()_i \mid {}_{\texttt{new}}()_i \mid (j)_i \mid .*$$

At runtime, values are associated with paths, and conversely, each path represents a set of values, as defined by recursion on the structure of paths:

- $\varepsilon$: the empty path $\varepsilon$ denotes any value that is not used for modeling the library API.
- $\texttt{require}(n)$: the objects returned by the `require` function when passing the library name $n$ as argument.
- $p.n$: the values obtained when accessing a property of name $n$ of an object denoted by the path $p$.
- $p()_i$: the values returned by calling a function denoted by the path $p$ when the function is called with $i$ arguments.
- $p_{\texttt{new}}()_i$: the values returned by calling a function denoted by the path $p$ as a constructor with $i$ arguments (i.e., using the `new` keyword in the call).
- $p(j)_i$: the values of the $j$'th argument passed to a function that is denoted by the path $p$ and called with $i$ arguments (similarly, $p_{\texttt{new}}(j)_i$ represents values of constructor arguments).
- $p.*$: the elements of the arrays denoted by $p$.

▶ **Example 2.** The following listing shows a client that uses a hypothetical library *twice*, which has a single method `twice` that takes an object and returns an object with the property `res` that contains the doubled value of the `x` property of the argument object.

```
19 // Twice library
20 module.exports.twice = function(t) {
21   return { res : t.x * 2 };
22 }
```

```
23 // Client
24 var m = require("twice");
25 var a = {x: 42, y: 43};
26 var b = m.twice(a);
27 a.y + b.res;
```

Consider which parts of the *twice* library are part of its public API. The value of the property `b.res` must be part of the public API, because `b` is coming from the library and `res` is accessed by the client on line 27. Intuitively, the client expects the `res` property to be available on the return value of the call on line 26. This value is given the path $\texttt{require}(\mathit{twice}).\texttt{f}()_1.\texttt{res}$. Likewise, the value of the property `a.x` is also part of the public API with the path $\texttt{require}(\mathit{twice}).\texttt{f}(\texttt{1})_1.\texttt{x}$. The `twice` method reads the `x` property, so it should be present on the argument passed to `twice`. The value of the property `a.y`, instead, is not part of the library API since it is never read by the `twice` method (even though `a` is passed to `twice`). We describe in detail our mechanism for distinguishing between public and private parts of the API in Section 5.3.

Note that a value can be given different paths during a single program execution, and multiple values can be denoted by the same path. We include the number of arguments in the function invocation action, to distinguish two invocations of the same function with different numbers of arguments. JavaScript developers often define variadic functions where the function behavior changes depending on the number of arguments. For example, consider the `map` function of *lodash*, which implements the standard higher-order map function. In *lodash*, the function implicitly chooses the identity function as its function argument if no other argument is supplied. Hence, it is beneficial for the precision of the API model to distinguish an invocation of `map` where a function argument is supplied from one where no function argument is supplied. We leave more complex forms of overloading to future work.

## 5.2 Types

We use a notion of types that extends the basic types of ECMAScript (strings, numbers, objects, etc.) with types that have a special meaning in Node.js, for example, arrays, sets, maps, event-emitters, and streams. We also include intersection types that are used to capture the prototype hierarchies of objects, as explained later. Union types are used to easily join different observations.

▶ **Definition 3** (Types). A *type* $t \in \textbf{Type}$ is a term in the following grammar:

$$t ::= \circ \mid b \qquad\qquad t \in \textbf{Type}$$

$$\begin{aligned} b ::= {} & b \vee b' \mid b \wedge b' \mid \texttt{undefined} \mid \texttt{string} \mid \texttt{boolean} \mid \texttt{number} \mid \texttt{object} \mid \texttt{function} \\ & \mid \texttt{array} \mid \texttt{set} \mid \texttt{map} \mid \texttt{event-emitter} \mid \texttt{stream} \mid \texttt{throws} \end{aligned}$$

To simplify the presentation, we only show a representative subset of the Node.js types. The type $b \vee b'$ is a union type, while $b \wedge b'$ is an intersection type. The type $\circ$ has a special use: If a path $p$ is ascribed the type $\circ$ in a model (i.e., $\pi(p) = \circ$), then no values represented by the path have been observed during client test execution, therefore they are assumed not to be part of the public interface of the library. (Note that the special type $\circ$ and the special path $\varepsilon$ are both used for identifying the library API; $\circ$ is used in the generated models, and $\varepsilon$ is used for tagging values in our instrumented interpreter as explained in Section 5.3.) The special type `throws` is ascribed to functions that throw exceptions. Considering exceptions as a special type is unusual, but as we demonstrate in Section 7, it fits our setting well.

The function $\textit{type}(v)$ gives the type of a runtime value $v$. The function assigns the corresponding type to primitive values, e.g., `string` to strings, but it does more for objects and functions to account for prototype inheritance: for example, the type of a function is $\texttt{function} \wedge \texttt{object}$ because functions are also objects, and similarly, the type of a set is $\texttt{set} \wedge \texttt{object}$.

Types do not distinguish between boxed and unboxed primitive values and do not represent function types explicitly by an arrow type as usually done in type systems. As explained above, the parameters and return types of a function are instead expressed as different paths.

As mentioned in Section 4, we use subtyping to detect changes in the public API of a library that are breaking. The intuition, according to the Liskov substitution principle, is that subtyping should satisfy substitutability. Informally, if $t'$ is a subtype of $t$, denoted $t' <: t$, then values of type $t$ may be replaced with values of type $t'$ without affecting the desirable properties of the program [21]. In our case, if a library method returns a value of type $t$ in the old version and a value of type $t'$ in the new version where $t' <: t$, then behavioral subtyping tells us that there is no type-related breaking change.

JavaScript is a dynamic languages, with many different programming styles used by library developers, and there is no canonical notion of subtyping that perfectly fits all styles. This problem arises also in optionally typed languages, such as TypeScript, where the type checker has more than 10 different options that library developers can customize to better match their programming styles.[8] The subtyping relation that we use in our implementation is defined below, although we envision that library developers may want to customize some of the rules when adopting the type regression testing technique.

▶ **Definition 4** (Subtyping)**.** The subtyping relation $<:$ among types is the relation given by the reflexive, transitive closure of the following rules.

$$b <: b \vee b' \qquad b' <: b \vee b' \qquad b \wedge b' <: b \qquad b \wedge b' <: b'$$

$$\frac{b <: b'' \qquad b' <: b''}{b \vee b' <: b''} \qquad \frac{b'' <: b \qquad b'' <: b'}{b'' <: b \wedge b'} \qquad \texttt{object} <: \texttt{undefined} \qquad t <: \circ$$

The rules for intersection and union types are standard. Note that the subtyping relation, because of union and intersection types, is not anti-symmetric [15]: for example $b \wedge b' <: b' <: b \wedge b'$ whenever $b' <: b$. Consequently, types are not an order under the subtyping but only a preorder. Therefore, the least upper bound (also called join), which is needed for inference and checking, is not unique. For this reason, we implicitly work on the quotient order constructed from the preorder whenever we use the join operator $\sqcup$. The rule `object` $<:$ `undefined` is motivated by the following example, and $t <: \circ$ is relevant for Example 9.

▶ **Example 5.** Consider the patch update of the *express* library from 3.0.1 to 3.0.2:

```
// express 3.0.1
app.use = function(route, fn){
  ...
  return this._router.route.apply(this._router, args);
}
// express 3.0.2
app.use = function(route, fn){
  ...
  this._router.route.apply(this._router, args);
  return this;
}
```

The return type of `get` changes from `undefined` (the value returned by the `route` function is `undefined`) to `object` $\wedge$ `function` (the type of the `this` value). The reason the developers of *express* introduced this modification was to enable cascading of method calls, i.e., the ability to write `app.use(...).use(...)`. This update is clearly non-breaking, and the subtyping rule for `undefined` ensures that type regression testing does not consider this as a breaking change for *express* because `function` $\wedge$ `object` $<:$ `object` $<:$ `undefined`.

Note that, although unlikely, it is possible to write a client that relies on the fact that `use` returns `undefined` rather than a function, so that some library developers might prefer to be warned about a possible breaking change in this case. The simple example shows that it may be worthwhile to allow JavaScript developers to customize the subtyping rules.

[8] https://www.typescriptlang.org/docs/handbook/compiler-options.html

## 5.3 Instrumented Interpreter

We will explain our approach for public API discovery though the rules of an instrumented interpreter on a subset of the JavaScript language. At the beginning of the execution of the client tests, the model $\pi$ assigns the type $\circ$ to all paths. As values are determined to be part of the public API, the corresponding path is assigned a new type using the *type* function. Eventually, at the end of the test execution, $\pi$ will hold a model of the public API of the library, corresponding to the subset that the client has used in the tests.

It is important to notice that the model which we build may not exactly match what the library developer intended as the public API. For example, if the library developer states in the documentation that clients are not supposed to read a certain value, but some client does it anyway, then the value will be considered part of the public API. Without a formal and generally accepted mechanism for specifying and enforcing encapsulation, any client usage taking place in practice has to be regarded as legitimate.

We describe the evaluation of a representative subset of JavaScript language constructs in A-normal form, i.e., where every subterm has been evaluated to a value [11]. For simplicity, we focus on a core language and do not explain the instrumentation of binary operators, constructor invocations, writes to local variables, exceptions, functions with multiple parameters, etc. The purpose of the following definitions is to explain the API discovery mechanism as an instrumentation of an existing JavaScript interpreter; in particular we are here not interested in all the details of JavaScript semantics, which are described elsewhere [14, 6].

▶ **Definition 6** (A-Normal Forms). The A-normal forms considered are the ones below; $v$ denotes values, and $c$ denotes constants.

$$
\begin{array}{rlr@{\qquad\qquad}l}
e ::= & c & \text{(literal constant)} & e \in \mathbf{Exp} \\
& \mid x & \text{(variable read)} & \\
& \mid v\,[v] & \text{(property access)} & \\
& \mid v\,[v] = v & \text{(property update)} & \\
& \mid v\,(v) & \text{(function call)} &
\end{array}
$$

The instrumented interpreter is defined by a big-step operational semantics. We assume that the original semantics uses judgments of the form $\langle e, \rho, \sigma \rangle \Downarrow \langle v, \sigma' \rangle$, which associate an initial term $e$, an environment $\rho$, and a store $\sigma$ with a resulting value $v$ and a store $\sigma'$. The environment $\rho$ maps variable identifiers to primitive values (strings, numbers, booleans, and undefined) or to store locations $l$. The store $\sigma$ maps locations to objects and closures. Objects are similar to environments, mapping identifiers to primitive values or to store locations.

The instrumented semantics $\langle e, \rho, \sigma, \pi \rangle \Downarrow^I \langle v, \sigma', \pi' \rangle$ extends the semantics with the $\pi, \pi'$ components where $\pi'$ is the computed public API model. The initial model $\pi$, used at the beginning of the program execution, associates the type $\circ$ with each path $p$. Similarly to information flow analyses, the instrumented interpreter uses tagged values $v^p$ where $p \in \mathbf{Path}$ is a path, in place of the original values [5]. Values $v^p$ and $v^{p'}$ are indistinguishable by JavaScript programs for any $p, p'$. Recall that $\varepsilon$ is called the empty path and is intended to represent values that are not part of the public API.

The key idea about the use of tagged values is to preserve the following property: If a value $v^p$ is read in the client code and has a nonempty path $p$, then the value has been retrieved through the use of the public library API. Vice versa, if a value $v^p$ in the library code has a nonempty path, then the value has been passed to the library through the public library API.

Our instrumentation ensures that values that are passed between library and client code through the public API are always assigned a nonempty path. Initially, the only value with a nonempty path is the value returned by `require` when the library is first loaded.

▶ **Example 7.** Suppose the library *foo* is used by a client as follows:

```
var lib = require(foo);
var x = new lib.c();
lib.m(x);
```

The function values of `lib.m` and `lib.c` are tagged with $\texttt{require}(\mathit{foo}).\texttt{m}$ and $\texttt{require}(\mathit{foo}).\texttt{c}$, respectively. The object value of `x` is initially assigned the path $\texttt{require}(\mathit{foo}).\texttt{c}_{\texttt{new}}()_0$ because it is a value returned by the library constructor `c`. Assume the object has properties that are written by `c` and read by `m`, and are not accessed by the client code. Such properties should be considered private to the library, so that changes to them in a library update are not treated as type regressions. This scenario is detected by our instrumented interpreter by observing that `x` already has a nonempty path when it is passed to `m`. Therefore, the path of `x` is emptied (set to $\varepsilon$) to avoid the object being considered part of the public API.

In general, whenever a value crosses the API a second time, the path of that value is set to $\varepsilon$, so that its usages are not recorded as being part of the public API.

The semantic rules for the instrumented interpreter are shown in Figure 1. We use the notation $\{p \mapsto t\}$ meaning the model that maps the path $p$ to the type $t$ and all other paths to $\circ$. The join $\pi \sqcup \pi'$ of two models $\pi, \pi'$ is the pointwise join of the types for each path. By a slight abuse of notation, when joining the type $\circ$ with another type we assume that it behaves as the unit, i.e. if $\pi(p) = \circ$ and $\pi'(p) = t$ then $(\pi \sqcup \pi')(p) = t$.

When evaluating a constant, it is tagged by the empty path (rule CONST). When evaluating a variable, the tagged value is looked up in the environment (rule VAR). The model is unaffected in both cases.

At property accesses, the value of a property needs to be retrieved from a target object, and the model should be updated to reflect the type of the property accessed whenever the target object is part of the public library API. The rule ACCESS uses an auxiliary function **Lookup** to handle the actual lookup of the property, possibly walking the prototype chain, to retrieve the value $v_r^{p_r}$ of the property $v_f^{p_f}$ of the object $v_t^{p_t}$ in the store $\sigma$. (We omit formal definitions of this and other auxiliary functions since they are not important for the tagging mechanism.) The rule distinguishes three cases for the path $p_r'$ that is used as tag for the resulting value $v_r$. Assume the property access occurs in the client code. In the first case, $p_t$ is nonempty and $p_r$ is empty, meaning that the object $v_t$ comes from the library, and the library code that wrote the property value $v_r$ to the object did not obtain that value from the client, so in this case the path $p_r'$ is set to $p_t\alpha$. The recorded action $\alpha$ is $.v_f$ where $v_f$ is the name of the property accessed, unless $v_t$ is an array, in which case the special array access action $.*$ is used. In the second case, $p_t$ and $p_r$ are both nonempty. This means that we are accessing a property of a library object, and the library code that wrote the property value to the object obtained that value from the client, so we are in a situation similar to Example 7, and we set $p_r'$ to $\varepsilon$ accordingly. In the third case, $p_t$ is empty, which means that the object $v_t$ comes from the client, so we simply keep the existing path $p_r$ for the resulting value. If the property access instead occurs in the library code, the reasoning is the same, but with the roles of client and library swapped. In either case, the $\pi$ component is updated to reflect that a value of type $\mathit{type}(v_r)$ has been observed for the path $p_r'$.

At property updates, a value is assigned as property on a target object. The rule UPDATE uses the auxiliary function **Update** to perform the actual update of the property $v_f^{p_f}$ of the

CONST

$$\langle c, \rho, \sigma, \pi \rangle \Downarrow^I \langle c^{\varepsilon}, \sigma, \pi \rangle$$

VAR

$$\frac{\rho(x) = v^p}{\langle x, \rho, \sigma, \pi \rangle \Downarrow^I \langle v^p, \sigma, \pi \rangle}$$

ACCESS

$$\frac{\begin{gathered}\mathbf{Lookup}(v_t^{p_t}, v_f^{p_f}, \sigma) = v_r^{p_r} \\ p'_r = \begin{cases} p_t\alpha & \text{if } p_t \neq \varepsilon \wedge p_r = \varepsilon \\ \varepsilon & \text{if } p_t \neq \varepsilon \wedge p_r \neq \varepsilon \\ p_r & \text{if } p_t = \varepsilon \end{cases} \qquad \alpha = \begin{cases} .* & \text{if } v_t \text{ is an array} \\ .v_f & \text{otherwise} \end{cases} \\ \pi' = \pi \sqcup \{p'_r \mapsto \mathit{type}(v_r)\}\end{gathered}}{\langle v_t^{p_t}\,\texttt{[}v_f^{p_f}\texttt{]}, \rho, \sigma, \pi \rangle \Downarrow^I \langle v_r^{p'_r}, \sigma, \pi' \rangle}$$

UPDATE

$$\frac{\begin{gathered} p'_a = \begin{cases} \varepsilon & \text{if } p_t \neq \varepsilon \\ p_a & \text{if } p_t = \varepsilon \end{cases} \\ \mathbf{Update}(v_t^{p_t}, v_f^{p_f}, v_a^{p'_a}, \sigma) = \sigma' \end{gathered}}{\langle v_t^{p_t}\,\texttt{[}v_f^{p_f}\texttt{]} = v_a^{p_a}, \rho, \sigma, \pi \rangle \Downarrow^I \langle v_a^{p'_a}, \sigma', \pi \rangle}$$

CALL

$$\frac{\begin{gathered} p'_a = \begin{cases} p_f(1)_1 & \text{if } p_f \neq \varepsilon \wedge p_a = \varepsilon \\ \varepsilon & \text{if } p_f \neq \varepsilon \wedge p_a \neq \varepsilon \\ p_a & \text{if } p_f = \varepsilon \end{cases} \\ \mathbf{Call}(v_f^{p_f}, v_a^{p'_a}, \sigma, \pi) = \langle v_r^{p_r}, \sigma', \pi' \rangle \\ p'_r = \begin{cases} \texttt{require}(v_a) & \text{if } v_f = l_{\text{require}} \\ p_f()_1 & \text{if } v_f \neq l_{\text{require}} \wedge p_f \neq \varepsilon \wedge p_r = \varepsilon \\ \varepsilon & \text{if } v_f \neq l_{\text{require}} \wedge p_f \neq \varepsilon \wedge p_r \neq \varepsilon \\ p_r & \text{if } v_f \neq l_{\text{require}} \wedge p_f = \varepsilon \end{cases} \\ \pi'' = \pi' \sqcup \{p'_a \mapsto \mathit{type}(v_a)\} \sqcup \{p'_r \mapsto \mathit{type}(v_r)\} \end{gathered}}{\langle v_f^{p_f}\,\texttt{(}v_a^{p_a}\texttt{)}, \rho, \sigma, \pi \rangle \Downarrow^I \langle v_r^{p'_r}, \sigma', \pi'' \rangle}$$

**Figure 1** Semantic rules of the instrumented interpreter.

object $v_t^{p_t}$ with the value $v_a^{p'_a}$ in the store $\sigma$, resulting in a new store $\sigma'$ where the object property has been updated. The path $p'_a$ of the value being assigned is selected as follows. If the path $p_t$ of the object is nonempty, then we are intuitively sending a value across the boundary between client and library. If $p_a$ is also nonempty, it means that the value now crosses the API boundary a second time, so we set $p'_a$ to $\varepsilon$. In all other cases, we simply preserve the existing path $p_a$. (In particular, this means that in the situation where $p_t$ is nonempty and $p_a$ is empty, $p'_a$ becomes $\varepsilon$, which is the only sensible choice with our current language of dynamic access paths.)

The rule CALL models the instrumentation of function calls. We use the auxiliary function **Call** to perform the actual call to the function $v_f^{p_f}$ with argument $v_a^{p'_a}$, store $\sigma$, and initial model $\pi$. The argument value $v_a$ is tagged by the path $p'_a$, which is selected as follows. Assume, without loss of generality, that the call occurs in the client code. In the first case of the definition of $p'_a$, $p_f$ is nonempty and $p_a$ is empty, meaning that the function $v_f$ comes from the library and the argument value $v_a$ comes from the client. In this case, the path $p'_a$ is set to $p_f(1)_1$, indicating that $v_a$ is used as first argument to the library function $p_f$ when executing the function body. (This generalizes naturally to functions with multiple arguments.) The two remaining cases follow the same reasoning as for $p'_r$ in rule ACCESS. The auxiliary function **Call** returns a value $v_r^{p_r}$, a updated store $\sigma'$, and an updated model $\pi'$. The path $p'_r$ used to tag the resulting value $v_r$ is decided depending on the the function $v_f$, its path $p_f$, and the path $p_r$ of the returned value. If $v_t$ is the `require` function, written $v_t = l_{\text{require}}$, then the path $p'_r$ is $\texttt{require}(v_a)$. Otherwise, the path is either set to $\varepsilon$ (according to the principle already discussed according to which the values crossing the API boundary twice are given an empty path), $p_f()_1$ (if the call is to a library function and the resulting value came from the library side), or kept as $p_r$. The resulting model $\pi''$ collects the observations from the execution of the function and the types of the function argument and the resulting value.

As described above, the instrumented interpreter generates a model for each client test being run. A complete model for a library version is made by joining all the models of the client tests using the $\sqcup$ operator.

## 6 Type Regression Testing

Every mapping in a model $\pi$ establishes a sort of mutual expectation between the client and the library. The direction of such an expectation depends on the path structure. For example, if $\pi(\texttt{require}(\mathit{lib})) = t$, then it is the client that expects an object of type $t$ from the library *lib*. The same direction applies to properties accessed on the object returned from `require`. For example, if $\pi(\texttt{require}(\mathit{lib}).\texttt{p}) = t$, then the client expects that the library has a property `p` of type $t$. The direction of the expectation flips upon an argument action. For example, if $\pi(\texttt{require}(\mathit{lib}).\texttt{p}(j)_i) = t$, then it is the library that expects the client to pass a value of type $t$.

The direction of the expectation affects the direction of the subtyping to use when checking for type regressions in library updates, much like covariance and contravariance in standard type checking. For example, if $\pi(\texttt{require}(\mathit{lib})) = t$ in version 1.0.0 and $\pi'(\texttt{require}(\mathit{lib})) = t'$ in version 1.1.0, then we have to check that $t' <: t$. Symmetrically, if the type for $\pi(\texttt{require}(\mathit{lib})(j)_i) = t$ in version 1.0.0 and it is $\pi'(\texttt{require}(\mathit{lib})(j)_i) = t'$ in version 1.1.0, then we have to check that $t <: t'$ because the library is certainly allowed to relax its expectations on minor version upgrades but it cannot require a more specific type.

We use the relation $<:_p$ to ensure that the direction of the typing relation is correct. It is inductively defined on the structure of the path $p$. The direction is switched on every argument action, as with contravariance for traditional function subtyping. In the base case where the path is empty, the ordinary subtyping relation $<:$ is used:

$$\frac{t <: t'}{t <:_\varepsilon t'} \qquad \frac{\alpha = (j)_i \qquad t' <:_{p'} t}{t <:_{p'\alpha} t'} \qquad \frac{\alpha \neq (j)_i \qquad t <:_{p'} t'}{t <:_{p'\alpha} t'}$$

We can now define precisely how to detect type regressions. Note that type regressions are never reported for empty paths, because empty paths represent values that are not part of the public API of the library.

▶ **Definition 8** (Type Regression)**.** Let $\pi$ and $\pi'$ be models of an old and a new library version, respectively. A nonempty path $p$ exhibits a *type regression* whenever $\pi'(p) \not<:_p \pi(p)$.

▶ **Example 9.** Continuing the example from Section 3, the path where the type regression is detected is $p = \texttt{require}(\mathit{lodash}).\texttt{merge}(3)_4.\texttt{toString}$. In the old model $\pi(p) = \circ$, while in the new model $\pi'(p) = \texttt{function} \wedge \texttt{object}$. By definition, the path $p$ exhibits a type regression: $\texttt{function} \wedge \texttt{object} \not<:_p \circ$ because $\circ <: \texttt{function} \wedge \texttt{object}$ does not hold.

Note that if the opposite change was made to the library, such that `toString` is read in the old version of the library but not in the new version, then the rule $t <: \circ$ (see Definition 4) would prevent the type regression, as desired.

## 7 Evaluation

To evaluate the type regression testing technique, we developed a tool, NoRegrets.[9] The implementation consists of 1800 lines of TypeScript code and 6400 lines of Scala code. The TypeScript part implements the instrumentation described in Section 5, using ES6 proxies. The Scala part fetches test suites for the npm packages from GitHub and post-processes the generated models to detect type regressions as described in Section 6.

Our primary hypothesis is that the type regression testing technique can help a developer decide if an update should be marked as either a major update or as a minor or patch update. We test this hypothesis by considering the tool as a *binary classifier* [28, 29], that is, a decision procedure that, given an input (in our case, a library update), returns one of two possible outcomes (*breaking* or *non-breaking*). Our secondary hypothesis is that the tool improves the *dont-break* approach for finding breaking changes, by amplifying the ability of the client test suites to reveal breaking changes, and by providing accurate and meaningful type regression reports. These hypotheses lead to the following research questions:

**RQ1** How accurate is NoRegrets in the classification of library updates as breaking or non-breaking?

**RQ2** How does NoRegrets compare with the *dont-break* approach? Specifically:

1. How many breaking changes does NoRegrets find, and how many of those cannot be detected by the *dont-break* approach?
2. How many of the type regressions reported by NoRegrets are spurious?
3. When a failure is detected, is it easy to locate the root cause?

**Benchmarks.** We randomly selected 12 among the most depended upon libraries from npm[10] as benchmarks, listed in Table 1. Since the development and release process of those libraries is usually under scrutiny of many skilled developers, they can be considered high-quality libraries: their changelogs are usually accurate, minor updates rarely introduce breaking changes, and major updates are usually reserved for those situations where breaking changes have been introduced.

[9] **no**de.js type **regre**ssion **t**e**s**ter
[10] `https://www.npmjs.com/browse/depended`

**Table 1** Node.js libraries used in the experimental evaluation.

| Benchmark | LOC | Minor/Patch | Major | Client Test Suites | Model Size |
|---|---|---|---|---|---|
| *debug* 2.0.0 | 226 | 19 | 1 | 63 | 33 |
| *async* 2.0.0 | 1682 | 5 | 0 | 64 | 3316 |
| *lodash* 3.0.0 | 5225 | 16 | 1 | 42 | 4661 |
| *moment* 2.0.0 | 1041 | 31 | 0 | 5 | 5 |
| *express* 3.0.0 | 1011 | 95 | 1 | 4 | 54 |
| *chalk* 1.0.0 | 169 | 4 | 0 | 93 | 105 |
| *bluebird* 3.0.0 | 4827 | 29 | 0 | 16 | 503 |
| *react* 15.0.0 | 41 685 | 11 | 1 | 5 | 31 |
| *commander* 2.0.0 | 370 | 12 | 0 | 4 | 7 |
| *request* 2.0.0 | 626 | 98 | 0 | 9 | 13 |
| *body-parser* 1.0.0 | 89 | 55 | 0 | 3 | 6 |
| *q* 1.0.0 | 1152 | 9 | 1 | 8 | 277 |

For our experiments, we picked a recent major version of each library, together with all minor and patch updates up to the next major release. The main focus of these experiments is on minor and patch updates, which are the ones where the developers do not expect breaking changes, but we also include a few major updates to check that NoRegrets is able to classify those as breaking.

Table 1 contains additional details on the selected libraries: the name and the major version of the library, the number of lines of code in the major version of the library, the number of minor/patch and major updates of the library considered, the number of clients with test suites, and the size of the public API (counted as the number of non-$\circ$ paths in the inferred model, averaged over the different library versions). All the data collected comes from a snapshot of the npm repository taken in April 2017. To simplify the experiments we only consider clients that use the Mocha testing framework, and to reduce noise we omit test suites that do not succeed consistently on major releases.

Our open-source implementation of NoRegrets and all benchmarks and experimental data are available at `http://brics.dk/noregrets`.

## RQ1 (accuracy as binary classifier)

Since the benchmarks selected in this experiment are high-quality libraries, we assume that most of the minor and patch updates of our benchmarks are not introducing breaking changes, and that most of the major updates are introducing breaking changes. In this way, we can evaluate our tool by checking that it correctly classifies major and non-major updates as breaking and non-breaking, respectively.

NoRegrets reports type regressions for only 36 out of 384 minor or patch updates, and for 4 of the 5 major updates. A few of the type regressions detected at minor updates are actual breaking changes being introduced by mistake, for instance the one shown in the motivating example and all the ones discussed for RQ2.1 later in this section. Moreover, NoRegrets is in fact correct also for the major update that is not being reported: a manual inspection confirms that the update of *debug* to version 3.0.0 does not introduce any type-related breaking changes apart from removing functions that were already deprecated and therefore not used by any available clients. Even if we disregard the fact that some of the minor updates are actually breaking and some major updates are non-breaking, NoRegrets is able to give accurate suggestions to library developers: *In at least 90% of the cases,* NoRegrets *is able to correctly classify a library update as either major or as minor or patch.*

**Table 2** Breaking changes found.

| Benchmark | Changelog | Test Failure | Synthetic Client |
|---|---|---|---|
| *debug* 2.0.0 | 0 | 1 | 0 |
| *async* 2.0.0 | 0 | 3 | 0 |
| *lodash* 3.0.0 | 2 | 0 | 0 |
| *moment* 2.0.0 | 2 | 0 | 0 |
| *express* 3.0.0 | 0 | 0 | 18 |
| **Total** | 26 | | |

## RQ2 (comparison with the *dont-break* approach)

To answer the second research question, we manually inspected each type regression reported by NoRegrets on minor and patch updates. To reduce the time spent, we focused on 5 benchmarks (*debug*, *async*, *lodash*, *moment*, and *express*).

**RQ2.1.** We consider a type regression on a path as introducing a breaking change whenever (i) the developers reference the change in the changelog, (ii) the breaking change can be witnessed by a test failure of one of the selected clients, or (iii) we can construct a synthetic client that crashes because of the change. If none of these conditions are satisfied, then the type regression is classified as a false positive. The breaking changes in the second category are the only ones that can be identified by the *dont-break* approach. The synthetic clients in the third category may not be representative of typical clients, but they nevertheless witness breaking changes. Also, as demonstrated by our motivating example and by Example 10, breaking changes often cause problems exactly because clients use libraries in ways that the library developers did not anticipate.

▶ **Example 10.** An example of a breaking change in the first category is the one introduced by *moment* 2.5.1 where the library started using the method `hasOwnProperty`. Unfortunately, the method is only available on non-host objects in older browsers. It took until version 2.8.2 for developers to realize this fact and fix the problem.[11]

The main results of our inspection of the reported regressions are shown in Table 2. The Changelog column contains the number of breaking changes that are confirmed by the changelog, and the Test Failure and Synthetic Client columns show how many are witnessed by a failing preexisting test or a synthetic client, respectively. *Remarkably,* NoRegrets *is able to find 26 breaking changes in minor updates of high-quality libraries.* It is also notable that this is accomplished with few client tests; for example, the two breaking changes in *moment* are detected using only 5 client test suites.

Moreover, *only 4 of the breaking changes that we have found could also be identified by a test failure*, demonstrating that our approach amplifies client tests compared to the *dont-break* approach. Going back to Example 10, note that detecting the breaking change by the *dont-break* approach would not just require a test that triggers the invocation of `hasOwnProperty` on non-host objects, but the test should also be run in a specific browser. Instead, NoRegrets reports a type regression indicating that the object passed to *moment* should have a function `hasOwnProperty`.

[11] `https://github.com/moment/moment/pull/1874`

**RQ2.2.** On the 5 benchmarks, NoRegrets reports a total of 168 type regressions across 167 library updates. By manually inspecting these 168 reports we find that 96 indicate actual breaking changes. (Some reports have the same root cause, which is how we arrive at a total of 26 breaking changes.) Thus, the number of false positives is acceptable: on average around one warning is reported per library update, and the majority of the warnings indicate actual breaking changes. Moreover, in the situations where multiple warnings are reported at a library update, we find that investigating the cause of one warning often quickly shows that other warnings have the same cause and therefore can be dismissed with little effort.

▶ **Example 11.** In *lodash* 3.10.0, the developers changed the behavior of the public method `isPlainObject` to use a different heuristic for recognizing so-called plain objects. Quoting from the implementation: "In most environments an object's own properties are iterated before its inherited properties". The method was changed accordingly to inspect all properties of the given object using a for-in loop, which causes 44 type regressions to be reported, one for each property of the objects being passed to `isPlainObject`. We classified this case as a false positive: the type regressions do not identify a breaking change since the property values are not used for anything but equality checks. We only had to inspect one library code location to understand that the other type regression reports had the same cause.

In our inspection of the regression reports, we proceeded hierarchically by the length of the paths involved in type regressions. By doing so, in our classification, we only needed to inspect a total of 36 type regressions out of 168, to identify the root cause of the breaking change they were referring to or discard them as false positives.

As already mentioned, 26 of our investigations resulted in the identification of an actual breaking change. To give some examples, the type regressions for 5 of these 26 breaking changes are listed in the first first 5 rows of Table 3. The *lodash* and *moment* examples have already been explained in Section 3 and in Example 10. The *debug* example, which shows the type regression for the breaking change mentioned in Section 1, and the *async* example both involve the special `throws` type. In the *async* example, the call to `require` returns an object in version 2.0.1 but throws an exception in version 2.1.0 because the module has been moved. Notice that none of type rules in Section 5.2 explicitly involve the `throws` type. Thereby, it is a type error if a function either starts throwing exceptions or stops throwing exceptions after a library update, which the *async* example motivates well; moving a public module clearly affects the public API. Likewise in the *debug* example, a spelling error results in an excepting being thrown when the *debug* module is loaded, which breaks the public API. The *express* example is similar to the *moment* and *lodash* examples: the property `readable` of the function argument is not accessed in version 3.14.0 of *express*, but it is accessed in version 3.15.0, demonstrating that the type signature of the library function has changed.

NoRegrets also reported 72 type regressions that we categorized as false positives. Using the same approach as when investigating the true positives, we found that these false positives had 10 separate causes. Many of these were due to technical limitations of NoRegrets rather than of the type regression testing technique itself. ES6 introduces default exports of objects and functions, where the default exports are automatically read by Node.js when reading properties of the required object, but from the perspective of dynamic access paths it still looks like the properties are read from the default object, e.g., `require(`*foo*`).default.p` instead of just `require(`*foo*`).p`. In the 2.1.2 patch update of the *async* library, it started to use default exports resulting in 3 reports about breaking changes that are false positives. These are only false positives because a fallback mechanism is included to handle old installations that do not support default exports. Therefore, it is fair to assume that an inconsiderate developer, who did not include a fallback mechanism, may still have benefited from these warnings.

**Table 3** Type regression examples.

| Library Update | Path | Type Regression |
|---|---|---|
| *debug* 2.3.3 → 2.4.0 | $\texttt{require}(\mathit{debug})()_1$ | $\texttt{throws} \land \texttt{object} \not<: \texttt{function} \land \texttt{object}$ |
| *async* 2.0.1 → 2.1.0 | $\texttt{require}(\mathit{async/asyncify})$ | $\texttt{throws} \land \texttt{object} \not<: \texttt{function} \land \texttt{object}$ |
| *lodash* 3.2.0 → 3.3.0 | $\texttt{require}(\mathit{lodash}).\texttt{merge}(3)_4.\texttt{toString}$ | $\circ \not<: \texttt{undefined}$ |
| *moment* 3.5.0 → 3.5.1 | $\texttt{require}(\mathit{moment})(1)_1.\texttt{hasOwnProperty}$ | $\circ \not<: \texttt{function} \land \texttt{object}$ |
| *express* 3.14.0 → 3.15.0 | $\texttt{require}(\mathit{express})()_0(1)_2.\texttt{readable}$ | $\circ \not<: \texttt{boolean}$ |
| *lodash* 3.10.1 → 4.0.0 | $\texttt{require}(\mathit{lodash}).\texttt{pluck}$ | $\texttt{undefined} \not<: \texttt{function} \land \texttt{object}$ |
| *express* 3.21.2 → 4.0.0 | $\texttt{require}(\mathit{express}).\texttt{mime}$ | $\texttt{undefined} \not<: \texttt{object}$ |
| *lodash* 3.10.1 → 4.0.0 | $\texttt{require}(\mathit{lodash}).\texttt{forEach}(2)_3()_3$ | $\texttt{undefined} \not<: \texttt{throws} \land \texttt{object}$ |

Another cause of false positives is that `hasOwnProperty` is not being instrumented, which is due to limitations of ES6 proxies.

**RQ2.3.** For each breaking change detected by NoRegrets, the type regression report contains the involved dynamic access path $p$ and associated types $\pi(p)$ and $\pi'(p)$. In contrast, when the *dont-break* approach detects a breaking change, it only provides the failing client test, with no information about the interactions between the client code and the library.

Based on our experience with the *dont-break* approach in the preliminary study (Section 2) and the NoRegrets approach in the experiment for RQ2.1, we find that type regression reports greatly simplify the investigations of the breaking changes. The library developer does not need to understand what the client tests are doing, and can focus exclusively on the changes in the library's codebase that have resulted in the changes of the public API. Since NoRegrets additionally records the call-stack at the point when a new type observation is created, dynamic access paths can easily be correlated with actions performed deep down in the private code of the library. A typical example is the one discussed in Section 3 where the path $\texttt{require}(\mathit{lodash}).\texttt{merge}(3)_4.\texttt{toString}$ shows that the coercion performed in the private `isIterateeCall` function is actually performed on the third argument of the `merge` function, and such information is not available if using the *dont-break* approach.

The last three rows of Table 3 show examples of type regressions found by NoRegrets in major updates. The first two examples show that the `pluck` function was removed from *lodash* in version 4.0.0 and that the `mime` property was removed from *express* in version 4.0.0. The last example, involving the `forEach` function of *lodash*, is a little more subtle. The `forEach` function takes 3 parameters in version 3.10.1, a collection, a callback function, which is applied to each element in the collection, and an object that is used as the `this` object in the callback. However, in version 4.0.0, the ability to set the `this` object of the callback is removed from the `forEach` function. After the update, reading a property of `this` in the callback causes a type error since `this` is now `undefined`.

## Discussion

The experimental evaluation is based on relatively few client test suites, which limits the fraction of the public APIs that are modeled and thereby reduces NoRegrets's ability to detect breaking changes. The libraries used in the experiments have thousands of clients, but our current implementation uses a fairly simply technique to locate clients and retrieve their test suites. In particular, it currently does not look for clients on GitHub but only uses npm. Also, tests are rarely published together with packages on npm, so NoRegrets requires that the packages.json file for each client contains a URL to a GitHub repository and that this repository contains a git tag matching the client version that is required. Many clients do not include tags in their repositories, so we chose to discard those clients. In principle, this issue could be alleviated by comparing the source code in all the repository commits with the source of the client published on npm, but we leave that to future work. Additionally, as mentioned we exclude test suites that do not succeed consistently on major releases. This is a well-known problem: In a recent study of 373 popular JavaScript applications, 41 of the packages had tests that failed or froze, and 3 had build or deployment issues [10]. This problem could in principle be mitigated by using a more fine-grained approach where NoRegrets looks at test failures at the granularity of single tests rather than entire test suites. Furthermore, the ES6 proxy mechanism used by NoRegrets sometimes interferes with tests causing them to fail, so we have to disregard those too to avoid noise. This is a known problem with opaque ES6 proxies, which has already been addressed by Thiemann et al. [17]. We could similarly solve this problem by modifying Node.js such that proxies becomes transparent, but this is again a technical limitation of our current implementation that could be alleviated with further implementation work. Still, the experimental results obtained with our current proof-of-concept implementation suffice to demonstrate the potential of the type regression testing idea.

Another opportunity for improvements is to investigate extensions of our notion of types that could arguably enable NoRegrets to better fit specific programming constructs. For example, Andreasen et al. [3] show that parametric polymorphism and recursive types could be beneficial to type JavaScript functional programming constructs used in practice. Other possible extensions include representations of tuple types, polymorphic functions, and variadic parameters. Although these are theoretically interesting ideas, and could easily be implemented in NoRegrets, in our evaluation we have not yet encountered concrete use-cases to justify the technical effort to introduce them.

## 8 Related Work

**Studies of npm.** Several experimental studies have investigated the npm repository [10, 19]. A study on JavaScript repositories showed that regression testing is a common practice, with an average of 78% of the packages having at least one test [10]. Two studies focus on the structure of the npm dependency network [30, 18]. In one of the studies, it is shown that the mean number of direct dependencies is 6, and that this number seems to be growing rapidly [30]. The same study also showed that the percentage of packages that are depended upon by other packages is only 27.5%, and a few popular packages are widely used by other packages. This should not be seen as a threat to the general applicability of our technique. If a package has no dependencies, then it matters little that the packages developer adheres to the semantic versioning principle. Another study has shown that the number of transitive dependencies is 10 times the number of direct dependencies and confirms that the number of dependencies of packages is growing exponentially, with a 60% increase in 2016 [18].

**Studies of library updates and breaking changes.** Our preliminary study is the first published study of the prevalence of breaking changes in the npm repository. So far, research on breaking changes has focused on other ecosystems, mainly Java. An experimental study by Derr et al. [9], involving 203 developers, analyzed the reasons behind many Android applications using outdated libraries versions. More than 50% of the participants indicated that one of the reasons was to "prevent incompatibilities". The authors developed a tool to compute the difference between the public API of two Java library versions, which they used to show that as many as 39% of minor and patch updates should have been flagged as major, which justifies the skepticism about the guarantees of semantic versioning expressed by the 203 developers.

The study from Raemakers et al. [26] addressed the use and misuse of semantic versioning in the Maven repository for Java packages. They conclude that "one third of all releases introduces at least one breaking change, and that this figure is the same for minor and major releases, indicating that version numbers do not provide developers with information in stability of interfaces", showing that breaking changes are prevalent in Maven repositories. A similar study by Jezek et al. [16] on 109 Java open-source libraries discovered that every library introduces at least one breaking change of the public API in non-major updates.

Other studies are concerning the relation between library updates and breaking changes for JavaScript libraries. Mirhosseini and Parnin [22] showed that breaking changes, understanding the implications of changes, and migration effort are among the top concerns of JavaScript developers. A small user study among npm package maintainers showed that package updates are mostly coordinated by personal communications between developers [7]. A follow-up study, comparing 8 npm library developers to Eclipse and R/CRAN developers, showed that "npm developers were more willing than developers of other platforms to perform breaking changes in the name of progress" [8]. A study of the prevalence of client side vulnerabilities in web applications also showed that, like in the Maven system, many applications are using outdated libraries [20]. Zerouali et al. also found that many dependencies in the npm system are outdated due to too strict version constraints, and conclude that developers are reluctant to update dependencies since they want to avoid incompatible changes [31]. Wittern et al. [30] showed that 29% of all package.json version constraints specify a fixed version, while 68% of the constraints allow either all minor and patch updates or just all patch updates. The remaining 3% are free ranging constraints that also allow major updates.

**Type inference for dynamic languages.** Dynamic inference of types for dynamic languages is a widely studied topic [2, 3, 27, 1, 24]. However, this paper is the first one to also distinguish between private and public parts of a library's API. The use of runtime traces to learn types has already been exploited for Ruby [2], JavaScript [3, 27], and Dart [1]. One notable difference with our approach is that we do not ascribe types to syntactical elements of the programs, but instead to our notion of dynamic access paths. TypeDevil [24] uses a dynamic analysis to gather runtime type information where types are either primitive or records of types. Inconsistencies of the observed types are reported as potential bugs. Other forms of dynamic analysis for JavaScript are discussed in a recent survey [4].

**Detection of breaking changes.** The only other tool that also aims at detecting breaking changes in npm package updates is the *dont-break* tool. Unfortunately, we were not able to make *dont-break* work properly, but we applied the same methodology in the preliminary study as discussed in Sections 2 and 7.

Greenkeeper is a service that helps packages maintainers avoid introducing dependency updates that contain breaking changes.[12] Instead of using range-based dependency constraints that allow all minor and patch updates, packages that use Greenkeeper will fix each dependency to a specific version. Whenever a new version of a dependency is available, Greenkeeper will run the tests of the package with the updated dependency to verify that the update did not break anything.

[12] `https://greenkeeper.io/`

Java's binary compatibility conditions [13, Chapter 13] and tools like JAPICC [23] make it possible to automatically detect type-related breaking changes in Java libraries. A disciplined set of guidelines for upgrading library releases have also been developed within the IBM's System Object Model to guarantee binary compatibility [12].

## 9 Conclusion

We have shown that breaking changes do occur in minor and patch updates of npm packages and that the majority of the breaking changes are type-related. Furthermore, we have designed a novel technique called type regression testing that detects type-related breaking changes across library versions, by leveraging the test suites of the library's clients. Type regression testing uses an instrumented JavaScript interpreter to build a model of a library's API through dynamic observations of how the client tests interact with the library. The models use the notion of dynamic access paths to give types to the individual components of the library's API. Specific differences in the model across two library versions are identified as type regressions, indicating that a breaking change likely has occurred.

We have implemented type regression testing in the tool NoRegrets. Our evaluation shows that NoRegrets is capable of detecting 26 breaking changes in 167 minor and patch updates of 5 high quality npm packages, and most of those breaking changes could not have been detected by existing techniques. We also find that NoRegrets reports only a small number of false positives, and that the reported type regressions make it easy for the developer to determine the causes of the breaking changes. Furthermore, NoRegrets correctly classifies at least 90% of the updates as either major or as minor or patch.

## References

1 Christoffer Quist Adamsen, Gianluca Mezzetti, and Anders Møller. Analyzing test completeness for dynamic languages. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, ISSTA 2016, Saarbrücken, Germany, July 18-20, 2016*, pages 142–153, 2016. doi:10.1145/2931037.2931059.

2 Jong-hoon (David) An, Avik Chaudhuri, Jeffrey S. Foster, and Michael Hicks. Dynamic inference of static types for Ruby. In *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 459–472, 2011. doi:10.1145/1926385.1926437.

3 Esben Andreasen, Colin S. Gordon, Satish Chandra, Manu Sridharan, Frank Tip, and Koushik Sen. Trace typing: An approach for evaluating retrofitted type systems. In *30th European Conference on Object-Oriented Programming, ECOOP 2016, July 18-22, 2016, Rome, Italy*, pages 1:1–1:26, 2016. doi:10.4230/LIPIcs.ECOOP.2016.1.

4 Esben Andreasen, Anders Møller, Liang Gong, Michael Pradel, Marija Selakovic, Koushik Sen, and Cristian-Alexandru Staicu. A survey of dynamic analysis and test generation for JavaScript. *ACM Computing Surveys*, 50(5):66:1–66:36, 2017.

5 Thomas H. Austin, Tim Disney, Alan Jeffrey, and Cormac Flanagan. Dynamic information flow analysis for featherweight JavaScript. Technical Report UCSC-SOE-11-19, UC Santa Cruz, 2011.

6 Martin Bodin, Arthur Charguéraud, Daniele Filaretti, Philippa Gardner, Sergio Maffeis, Daiva Naudziuniene, Alan Schmitt, and Gareth Smith. A trusted mechanised JavaSript specification. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 87–100. ACM, 2014.

7 Christopher Bogart, Christian Kästner, and James D. Herbsleb. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *30th IEEE/ACM International Conference on Automated Software Engineering Workshops, ASE Workshops 2015, Lincoln, NE, USA, November 9-13, 2015*, pages 86–89, 2015. doi:10.1109/ASEW.2015.21.

8 Christopher Bogart, Christian Kästner, James D. Herbsleb, and Ferdian Thung. How to break an API: cost negotiation and community values in three software ecosystems. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18, 2016*, pages 109–120, 2016. doi:10.1145/2950290.2950325.

9 Erik Derr, Sven Bugiel, Sascha Fahl, Yasemin Acar, and Michael Backes. Keep me updated: An empirical study of third-party library updatability on Android. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017*, pages 2187–2200, 2017.

10 Amin Milani Fard and Ali Mesbah. JavaScript: The (un)covered parts. In *2017 IEEE International Conference on Software Testing, Verification and Validation, ICST 2017, Tokyo, Japan, March 13-17, 2017*, pages 230–240. IEEE Computer Society, 2017.

11 Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 237–247, 1993. doi:10.1145/155090.155113.

12 Ira R. Forman, Michael H. Conner, Scott Danforth, and Larry K. Raper. Release-to-release binary compatibility in SOM. In *OOPSLA'95, Proceedings of the Tenth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications, Austin, Texas, USA, October 15-19, 1995*, pages 426–438. ACM, 1995.

13 James Gosling, Bill Joy, Guy L. Steele, Gilad Bracha, and Alex Buckley. *The Java Language Specification, Java SE 8 Edition.* Addison-Wesley Professional, 1st edition, 2014.

14 Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *ECOOP 2010 - Object-Oriented Programming, 24th European Conference, Maribor, Slovenia, June 21-25, 2010. Proceedings*, pages 126–150, 2010.

15 Atsushi Igarashi and Hideshi Nagira. Union types for object-oriented programming. *Journal of Object Technology*, 6(2):47–68, 2007. OOPS Track at the 21st ACM Symposium on Applied Computing, SAC 2006. doi:10.5381/jot.2007.6.2.a3.

16 Kamil Jezek, Jens Dietrich, and Premek Brada. How Java APIs break - an empirical study. *Information & Software Technology*, 65:129–146, 2015. doi:10.1016/j.infsof.2015.02.014.

17 Matthias Keil, Sankha Narayan Guria, Andreas Schlegel, Manuel Geffken, and Peter Thiemann. Transparent object proxies in JavaScript. In *29th European Conference on Object-Oriented Programming, ECOOP 2015, July 5-10, 2015, Prague, Czech Republic*, pages 149–173, 2015. doi:10.4230/LIPIcs.ECOOP.2015.149.

18 Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *Proceedings of the 14th International Conference on Mining Software Repositories, MSR 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 102–112. IEEE Computer Society, 2017.

19 Raula Gaikovina Kula, Ali Ouni, Daniel M. Germán, and Katsuro Inoue. On the impact of micro-packages: An empirical study of the npm JavaScript ecosystem. *CoRR*, abs/1709.04638, 2017. arXiv:1709.04638.

20 Tobias Lauinger, Abdelberi Chaabane, Sajjad Arshad, William Robertson, Christo Wilson, and Engin Kirda. Thou shalt not depend on me: Analysing the use of outdated JavaScript

libraries on the web. *Proceedings of Network and Distributed System Security Symposium (NDSS)*, 2017.

21 Barbara Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Trans. Program. Lang. Syst.*, 16(6):1811–1841, 1994. `doi:10.1145/197320.197383`.

22 Samim Mirhosseini and Chris Parnin. Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering, ASE 2017, Urbana, IL, USA, October 30 - November 03, 2017*, pages 84–94, 2017. `doi:10.1109/ASE.2017.8115621`.

23 Andrey V. Ponomarenko and Vladimir V. Rubanov. Backward compatibility of software interfaces: Steps towards automatic verification. *Programming and Computer Software*, 38(5):257–267, 2012.

24 Michael Pradel, Parker Schuh, and Koushik Sen. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Florence, Italy, May 16-24, 2015, Volume 1*, pages 314–324, 2015. `doi:10.1109/ICSE.2015.51`.

25 Tom Preston-Werner. Semantic versioning 2.0.0. `http://semver.org/`.

26 Steven Raemaekers, Arie Van Deursen, and Joost Visser. Semantic versioning versus breaking changes: A study of the Maven repository. *Proceedings - 2014 14th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2014*, pages 215–224, 2014. `doi:10.1109/SCAM.2014.30`.

27 Claudiu Saftoiu, Arjun Guha, and Shriram Krishnamurthi. Runtime type-discovery for JavaScript. Technical Report Brown University CS-10-05, 2010.

28 Marina Sokolova and Guy Lapalme. A systematic analysis of performance measures for classification tasks. *Inf. Process. Manage.*, 45(4):427–437, 2009.

29 Sofia Visa, Brian Ramsay, Anca L. Ralescu, and Esther van der Knaap. Confusion matrix-based feature selection. In *Proceedings of the 22nd Midwest Artificial Intelligence and Cognitive Science Conference 2011, Cincinnati, Ohio, USA, April 16-17, 2011*, pages 120–127, 2011. URL: `http://ceur-ws.org/Vol-710/paper37.pdf`.

30 Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the JavaScript package ecosystem. In *Proceedings of the 13th International Conference on Mining Software Repositories, MSR 2016, Austin, TX, USA, May 14-22, 2016*, pages 351–361, 2016. `doi:10.1145/2901739.2901743`.

31 Ahmed Zerouali, Eleni Constantinou, Tom Mens, Gregorio Robles, and Jesús M. González-Barahona. An empirical analysis of technical lag in npm package dependencies. In *New Opportunities for Software Reuse - 17th International Conference, ICSR 2018, Madrid, Spain, May 21-23, 2018, Proceedings*, volume 10826 of *Lecture Notes in Computer Science*, pages 95–110. Springer, 2018.