More Effective JavaScript Breaking Change Detection via Dynamic Object Relation Graph

DEZHEN KONG, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China
JIAKUN LIU, School of Information Systems, Singapore Management University, Singapore
CHAO NI, School of Software Technology, Zhejiang University, China
DAVID LO, School of Information Systems, Singapore Management University, Singapore
LINGFENG BAO*[†], The State Key Laboratory of Blockchain and Data Security, Zhejiang University, China

JavaScript libraries are characterized by their widespread use, frequent code changes, and a high tolerance for backward incompatible changes. Awareness of such breaking changes can help developers adapt to version updates and avoid negative impacts. Several tools have been targeted to or can be used to detect breaking change detection in the JavaScript community. However, these tools detect breaking changes using different ways, and there are currently no systematic reviews of these approaches. From a preliminary study on popular JavaScript libraries, we find that existing approaches, including simple regression testing, model-based testing and type differencing cannot detect many breaking changes but can produce plenty of false positives. We discuss the reasons for missing breaking changes and producing false positives.

Based on the insights from our findings, we propose a new approach named Diagnose that iteratively constructs an object relation graph based on API exploration and forced execution-based type analysis. Diagnose then refine the graphs and reconstruct the graphs in the newer versions of the libraries to detect breaking changes. By evaluating approach on the same set of libraries used in the preliminary study, we find that Diagnose can detect much more breaking changes (60.2%) and produce fewer false positives. Therefore, Diagnose is suitable for practical use.

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

Additional Key Words and Phrases: JavaScript, Breaking Changes, NPM

ACM Reference Format:
Dezhen Kong, Jiakun Liu, Chao Ni, David Lo, and Lingfeng Bao. 2025. More Effective JavaScript Breaking Change Detection via Dynamic Object Relation Graph. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA103 (July 2025), 22 pages. https://doi.org/10.1145/3728980

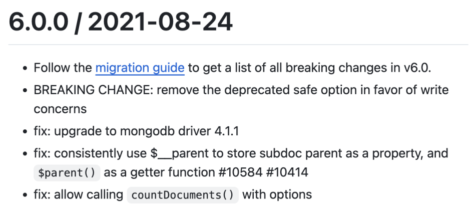
1 Introduction

JavaScript libraries are widely used in many programming fields [27, 29, 45], such as Web frontend, desktop applications, etc. The NPM registry hosts millions of JavaScript libraries. A key strength of the NPM ecosystem is its high tolerance for breaking changes. However, the NPM ecosystem

*Corresponding author.
[†]Also with Hangzhou High-Tech Zone (Binjiang) Blockchain and Data Security Research Institute.

Authors' Contact Information: Dezhen Kong, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, timkong@zju.edu.cn; Jiakun Liu, School of Information Systems, Singapore Management University, Singapore, Singapore, jkliu@smu.edu.sg; Chao Ni, School of Software Technology, Zhejiang University, Ningbo, China, chaoni@zju.edu.cn; David Lo, School of Information Systems, Singapore Management University, Singapore, Singapore, davidlo@smu.edu.sg; Lingfeng Bao, The State Key Laboratory of Blockchain and Data Security, Zhejiang University, Hangzhou, China, lingfengbao@zju.edu.cn.

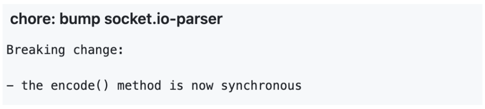
ACM 2994-970X/2025/7-ARTISSTA103
https://doi.org/10.1145/3728980



6.0.0 / 2021-08-24

- Follow the migration guide to get a list of all breaking changes in v6.0.
- BREAKING CHANGE: remove the deprecated safe option in favor of write concerns
- fix: upgrade to mongodb driver 4.1.1
- fix: consistently use $__parent to store subdoc parent as a property, and `$parent()` as a getter function #10584 #10414
- fix: allow calling `countDocuments()` with options

(a) A Breaking Change Documented in Changelog of Mongoose 6.0.0



chore: bump socket.io-parser

```
Breaking change:

- the encode() method is now synchronous
```

(b) A Breaking Changes Documented in Commit Message of Socket.io

Fig. 1. Examples of Documented Breaking Changes

recommends developers adhere to semantic versioning [42]. This means they should differentiate between version updates that introduce breaking changes and those that do not include incompatible code changes. In practice, developers can document breaking changes in *changelogs* and commit messages. For example, Figure 1a and 1b show the breaking changes documented in a *changelog* and a commit message, respectively.

To comply with semantic versioning [42], library developers should be aware of breaking changes in advance, and if downstream developers are informed about breaking changes as soon as possible, they can adapt to such version updates more smoothly. Therefore, there are various approaches available for detecting potential breaking changes in JavaScript libraries. However, the approaches for JavaScript breaking change detection are limited by their intrinsic design, For example, the regression testing based tools Dont-break [10], NoRegrets [27] (and NoRegrets+ [29]) are subject to insufficient test cases, and provide supports for a limited range of language specification. Additionally, in JavaScript, breaking changes can be complicated to analyze due to many JavaScript's dynamic nature (e.g., runtime module initialization, dynamic exportation [35]), and JavaScript-specific features (e.g., prototype chain, widely used higher order functions). Current approaches may not be able to deal with such complicated breaking changes.

To this end, we preliminarily evaluate the effectiveness of the aforementioned approaches, and a synthesized type differencing tool based on a TypeScript type inference tool [24] for detecting type-related breaking changes in 40 popular JavaScript libraries sampled from those with the highest number of dependents in the NPM registry. We find that simple regression testing tool Dont-break can hardly ever detect breaking changes. The model-based regression testing approach NoRegrets misses many breaking changes but produces false positives. Similarly, the synthesized type inference-based approach Tsinfer-node overlooks a proportion of breaking changes due to insufficient type analysis, while also producing a higher number of false positives. We also discuss the reasons for missing breaking changes and produce false positives, e.g., no sufficient client code, no enough API exploration, and inaccurate type analysis. Correspondingly, we provide insights for further improvements.

Based on the insights, we propose an automated approach named Diagnose (standing for Dynamic Object Relation Graph Generation for Node.js Libraries) to detect potential API changes. The overall process is iteratively constructing an object relation graph based on API exploration and forced execution-based type analysis. Diagnose then refines the graph by refining input value types, adding input from example code and pruning the graph. Specifically, Diagnose iteratively explores the object hierarchy of the imported library and adds nodes to the object relation graph. Through type analysis, Diagnose adds edges that represent the call relationship between nodes. And then the object relation graph will be refined to make types more accurate and reflect practical usage. Finally the breaking changes can be detected by reconstructing the dynamic object relation graph in the newer version of the library.

We also evaluate our approach on the 40 popular JavaScript libraries sampled from the NPM registry. The results show that Diagnose can detect much more breaking changes than existing tools, and report only 9 false positives. The results also illustrate that Diagnose can construct and reconstruct dynamic object relation graphs in a reasonable time, hence is more effective in practical use.

2 Background

2.1 Breaking Changes in JavaScript

With JavaScript libraries evolving and new versions released, incompatible changes are probably introduced. A number of studies have uncovered the general characteristics of breaking changes and their effects on the software ecosystem. For example, the research works [8, 40, 45] have demonstrated the widespread occurrence of breaking changes in the NPM ecosystem and their effects on client applications. Venturini et al. [45] found that in their sampled packages, 11.7% of all client packages and 13.9% of their releases are impacted by breaking changes. Notably, 44% of the breaking changes are introduced in minor or patch releases. However, according to Semantic Versioning [42], developers should not introduce breaking changes in non-major releases. Bogart et al. [1, 2] found that coarse-grained motivations for making breaking changes comprise requirements and context changes, bugs and new features, rippling effects from upstream changes, and technical debt from postponed changes.

The studies mentioned above illustrate the effects of breaking changes on the NPM ecosystem, but the detection of breaking changes is challenging. One of the challenges is due to the complicated syntactic features of JavaScript, making the analysis of JavaScript libraries more difficult. For example, unlike other statically typed languages, dynamic property access and native function calls in JavaScript are not simple to analyze [4]. Another challenge is about test suites in detecting breaking changes. In several prior works about breaking change detection [27, 46], researchers collected breaking changes by running downstream test cases: if a test case of a downstream project could not pass with a newer provider (after the code change), then the code change was identified as a breaking change. However, not all providers are dependent on many client applications. If a code change lacks a triggering test case in client applications, we cannot determine whether it is actually incompatible. On the other hand, client code may not follow the specification of providers and incorrectly access APIs, e.g., passing an improper value to a function, which will result in undefined behavior, including test failure, but it does not reflect a breaking change.

2.2 Existing Approaches for Breaking Change Detection

To the best of our knowledge, there are three types of tools can be utilized to detect breaking changes in JavaScript libraries. We provide a brief overview of them in this section.

Simple Regression Testing. The basic approach to detect breaking changes is to use regression testing. Specifically, if a test suite can pass under the old source code but fail under the new source code, it indicates that the new source code (after the code change) contains breaking changes. Dont-break [10] is a regression testing-based tool that verifies whether the test cases in dependent projects fail. If the current version of a library has some test cases that fail while the previous version does not, the current version may contain breaking changes. However, Dont-break needs manual configuration of dependent projects and can consume a significant amount of time in executing all test cases [29].

Model-based Regression Testing. Model-based testing is a testing technique that generates *models* that describe some behavior of a system. Mezzetti et al. proposed NoRegrets [27] and later Møller et al. enhanced it and proposed NoRegrets+ [29]. These two approaches generate models about dynamic access paths from the executions of client test code. Their core idea is that an API call should access the same set of properties and return the same type before and after the code change. Compared to simple regression testing, their approaches can capture more runtime-type information.

Example 1. Considering the following code:

```
let lib = require('foo');
let o = lib.func('bar');
assert(o.num === 0);
```

Then their approach can generate the following constraints:

(1) The call to require('foo').bar should return an object.
(2) The returned object of the call require('foo').bar() should contain a property num, which is a number.

The breaking changes can be discovered by analyzing the generated API models of two versions.

In contrast to NoRegrets, the major improvement of NoRegret+ is that it does not need to rerun all test cases and generate a new API model for the updated version of the library. However, sufficient test cases are still necessary to generate a good model. Since NoRegrets+ is designed to be the enhanced version of NoRegrets and can make use of more test cases, we only evaluate NoRegrets+ in our study.

Type Differencing. By comparing the types of the different versions of a JavaScript, we can straightforwardly know the potential breaking changes. Since TypeScript was proposed, JavaScript developers can write TypeScript declaration files (with the extension d.ts) for existing JavaScript libraries, without rewriting all JavaScript source code files using TypeScript. The GitHub repository DefinitelyTyped [9] hosts the TypeScript declaration files published by JavaScript developers, and modern IDEs can make use of type declaration files to intelligently indicate supported types when developers call the APIs in the JavaScript libraries.

Example 2. In the JavaScript library *joi*, client code can use the APIs as follows:

```
let schema = joi.string().uri({scheme : 'http'});
schema.validate(...);
```

Accordingly, developers can write the type declarations[1] shown in Figure 2. In this example, the return type of joi.string() corresponds to the interface StringSchema, and the type should have a property named uri (which is a function, accepting an argument of the type UriOptions). When the authors of *joi* would like to support a new option, they can add it to the interface UriOptions. After the type declarations are generated for the two consecutive versions of a library, type-related breaking changes can be obtained by comparing the generated type declaration files between the two versions.

There are several tools that can help generate such type declaration files, including Tsinfer [24], Dts-gen [11] and Dts-generator [6]. The tools are not designed for breaking changes, but can help us identify potential breaking changes with the similar criteria in contrast to NoRegrets+ (e.g., if an additional property is read, we consider it as a breaking change). Tsinfer is an automatic tool that generates TypeScript declaration files for a JavaScript library [24]. Tsinfer is built on top of

[1]See https://github.com/definitelyTyped/definitelyTyped/blob/master/types/hapi__joi/index.d.ts#L329.

```
interface StringSchema<TSchema = string> extends AnySchema<TSchema> {
  uri(options?: UriOptions): this;
  // more declarations
}

interface UriOptions {
  scheme?: string | RegExp | Array<string | RegExp>;
 // more declarations
}
```

Fig. 2. Type Declaration for *joi*

TSCHECK [14], based on heap snapshot analysis after the library is dynamically loaded into memory. It then gathers class hierarchy by exploration of the runtime snapshot and employs a static analysis for all functions in the snapshot. It also collects some information by dynamically calling some functions, e.g., the type of returned objects. The other tools, i.e., DTS-GEN and DTS-GENERATOR work similarly.

While the type declaration generation is not designed for type-related breaking change detection, we can compare the generated types before and after the version updates, using the similar criteria as NOREGRETS+, e.g., regarding the additional requirement of a property of an argument, and the removal of an exported object as breaking changes.

3 Preliminary Study

To illustratively understand the challenges in detecting breaking changes for JavaScript libraries, we present a preliminary study and discuss the key challenges in detecting such changes. We focus on the effectiveness of the approaches (Q1) and the reasons why these tools miss breaking changes (Q2) and produce false positives (Q3).

3.1 Experimental Settings

Library Selection. Given the vast number of JavaScript libraries in the whole NPM registry (over one million), and considering the unpopular libraries lack dependent projects (making them unsuitable for evaluating NOREGRETS+ and DONT-BREAK), we focused on identifying the top 1,000 JavaScript libraries with the highest number of clients. Similar to Møller et al.'s selection process [29], we randomly select 40 JavaScript libraries from the NPM registry ranked within 1,000 (we do not consider those that cannot be packaged into a CommonJS module, since we focus the libraries that can be used as a CommonJS module in the Node.js environment for compatibility). The selected libraries include some of well-known ones, e.g., *async*, *moment* (due to space limitation, we put the details of them into our replication package [23]). We ensure that all of the selected libraries have well-maintained changelogs, which assist us in identifying breaking changes acknowledged by developers.

Breaking Change Collection. Before evaluating the tools, we use the same method as adopted by Møller et al. [28] to identify the breaking change statements in software documentations, i.e., *changelogs* and commit messages. Specifically, they manually analyzed the documented breaking changes and used an API path access language to formally describe what APIs can be affected by a breaking change. Note that we do not consider the breaking changes that are not concerned with JavaScript source code, e.g., modifying *package.json* or dropping support for a specific version of Node.js. We manually identify 176 breaking changes in the 40 JavaScript libraries. Note that some

Table 1. Effectiveness of different Approaches on part of the benchmark libraries (due to space limitation, the detailed information is in the replication package [23])

Library	Dont-break		NoRegrets+		Tsinfer-node	
	Detected	FP	Detected	FP	Detected	FP
async	1/12	0	3/12	5	2/12	6
execa	0/13	0	6/13	4	6/13	4
jsonwebtoken	0/2	0	0/2	1	0/2	0
find-up	0/2	0	1/2	0	0/2	0
camelcase	0/0	0	0/0	0	0/0	0
moment	0/1	0	1/1	0	0/1	1
qs	0/0	1	0/0	2	0/0	2
path-to-regexp	0/0	0	0/0	2	0/0	3
has-flag	0/0	0	0/0	0	0/0	0
uuid	1/1	0	1/1	0	1/1	0
Not listed	7/145	0	7/145	25	5/145	30
Total	9/176	1	19/176	42	14/176	46

libraries do not have breaking changes, but we still retain them because they can help us evaluate whether the breaking change detection tools produce false positives.

Evaluation Settings. We here detail the evaluation settings for different approaches. Since Tsinfer-node loads the JavaScript code and monitors the runtime behavior in browsers, we re-implement to support Node.js libraries and ES6 features (the re-implemented version is called Tsinfer-node). For NoRegrets+, we also re-implement it using JavaScript to make it more configurable. Since NoRegrets+ takes the imported object by require function as a starting point, we package the ES6 modules into CommonJS modules using *rollup*[2] (since certain versions of some libraries are difficult to convert into CommonJS modules, we only consider the versions of libraries that can be successfully configured). When we run Dont-break and NoRegrets+, we follow the settings that is provided by Møller et al. [29], e.g., considering at most 2,000 clients for a benchmark library. For Tsinfer-node, we compare the generated function signatures from two versions and regard the signature changes as breaking changes.

Detection Result Analysis. After running the tools on our selected libraries, we collect the detected breaking change information (e.g., in a version, an export function's behavior is changed for certain test cases, note that different tools report different types of information), and check whether the reported information can match one of the breaking changes in our collected ones. If no information can match a breaking change that is documented in the changelog or commit message, we consider it as a *missed breaking change*, i.e., false negative. If the reported information cannot match any of our identified breaking changes, we say that it is a false positive. We manually and independently check whether the reported information actually related to breaking changes, and then we hold meetings to resolve the disagreements. We use thematic analysis steps recommended by Cruzes et al. [7] to summarize the reasons why the tools produce false positives and false negatives. Specifically, the authors write some phrases to describe each false-positive and false-negative, and obtain a series of phrases to describe a false positive or false negative case. Then, by clarifying each phrase's

[2]https://rollupjs.org

meaning, we can combine the phrases with similar meanings into one. Thus there is a collection of themes. Then, we try to cluster the similar themes (if possible). Finally, the themes are determined.

3.2 Result and Discussion

Q1: Effectiveness. We present the numbers of detected (missing) breaking changes and false positives of the approaches on part of the benchmark libraries in Table 1 for detailed analysis (the detailed numbers for each library are put in our replication package [23]). If an approach cannot be applied to a library, we consider the number of missing breaking changes as the total count of breaking changes in that library.

The simple regression testing tool DONT-BREAK can hardly ever detect breaking changes since there are often not sufficient client projects. NoRegrets+ can detect more breaking changes but produce a number of false positives. Similarly, Tsinfer-node also detects a small part of breaking changes while produces a number of false positives. To sum up, the current tools can detect a proportion of breaking changes but miss a large number of breaking changes, and may produce false positives.

Q2: Missed Breaking Changes. We here discuss why the existing approaches miss breaking changes.

- For Dont-break, the main reason is that the APIs in the JavaScript libraries are not accessed by client code. Even for the frequently used APIs, Dont-break may not detect the related breaking changes since the API change is subtle. For example, considering Logger constructor in *winston*, DONT-BREAK does not detect such a breaking change in 3.0.0: the handleExceptions property of the first argument of Logger is no longer supported,[3] since the client code does not use this property.
- For NoRegrets+, similar to Dont-break, NoRegrets+ also cannot detect the breaking changes related to APIs that are not frequently used. For example, for *async*, 6 of 12 identified breaking changes are related to the APIs not accessed by client code (e.g., async.memoize, async.doDuring). While for a part of changes in the APIs accessed by client code, NoRegrets+ can detect them, some breaking changes on these APIs cannot be detected, since they are related to subtle behavior change. For example, in *mongoose* 5.0.0, the property rawResult of the property options of Query is no longer supported, and many functions in Query.prototype no longer process the property passRawResult in the input argument named options[4]. However, since NoRegrets+ only regards the additional property read as a breaking change, hence this change is missed.
- For Tsinfer-node, the main reason is that Tsinfer-node does not effectively explore the APIs in a JavaScript library, hence it misses breaking changes in some APIs. The other reason is that the type analysis is not accurate, i.e., generating a lot of any and void types, hence there seem no type difference between the original and the updated APIs.

Q3: False Positives. We here discuss the reasons for false positives for the evaluated approaches.

- For NoRegrets+, the reason for generating false positives is that NoRegrets+ identifies all additional property reads as breaking changes (although in fact they are not all breaking). There are two typical example. In *mongoose* 6.0.0, the internal structure of the objects inherited from Document is refactored: the new property $__schema is added. The new API implementations will read the new property, then NoRegrets+ regards it as a breaking change.

[3]https://github.com/winstonjs/winston/commit/a470ab5
[4]https://github.com/automattic/mongoose/issues/5869

In *immutable* 4.0.0, the functions in the library may additionally read many internal properties like @@__IMMUTABLE_ITERABLE__@@ compared to version 3.x, and NoRegrets+ regards the additional reads when calling the collection-related APIs, e.g., immutable.Map, immutable.Seq, as breaking changes.

- For Tsinfer-node, the major reason for generating false positives is that Tsinfer-node is not aware native types of arguments (if there is no documentation about the function). It only generates an interface declaration for a string or array parameter. For example, if one parameter should be an array, Tsinfer-node infers that it should have a method push and a property length. If after the code change, another native function in Array.prototype is called, e.g., sort, Tsinfer-node will generate a different interface to represent the parameter specification. However, the argument is still an array. Therefore, the tool Tsinfer-node produces a lot of false positives.

Insights. From the above study, we can gain the following insights that guide the improvements of breaking change detection.

(1) With respect to the limitations of three approaches, the new detection approach should explore the APIs in a JavaScript library and systematically organize them.
(2) With respect to NoRegrets+ and Tsinfer-node, the property changes can not be directly identified as potential breaking changes since they are possibly for internal use.
(3) With respect to Tsinfer-node, during the analysis, such parameters should be correctly identified as native types.
(4) With respect to NoRegrets+ and Dont-break, we should not rely on the test suites too much since they are often not sufficient to produce behavior differences in regression testing.
(5) With respect to Tsinfer-node, the breaking change detection tool should employ more accurate type analysis.

4 Our Approach: Diagnose

Based on our insights from the preliminary study, we propose an approach named Diagnose that detects breaking changes with the help of the dynamic object relation graph. In each iteration, it adds nodes from object exploration and the results of type analysis. In Section 4.2, we describe the dynamic object relation graph and the overall procedure of how to build such a graph. In Section 4.1 and 4.3, we provide the details of type analysis and refinement with example code, which are the critical processes in the overall algorithm.

4.1 Forced Execution-based Enhanced Type Analysis

The type analysis is based on a forced execution framework [16, 21]. The original purpose of forced execution is to make the program continually run rather than interrupted by runtime errors. However, Diagnose mainly uses forced execution to record the possible property information during the execution and explore possible input types by multiple forced execution runs. Compared with type differencing tool Tsinfer-node, Diagnose utilize forced execution to generate effective input values rather than simply calling the functions (cf. insights 4 and 5).

Our forced execution focuses on the parameters of accessible functions. Similarly to previous works [16, 21], our basic idea is to set a fake value for arguments into a function since their type information is not determined, and the properties of fake values will be specified during the execution. We achieve this by recording operations on those fake values. For example, when an argument is an operand in an arithmetic operation, it is coerced to a number, and thus the argument can be determined to be a number, and the result of the operation is a fake number. When the property prop of the argument (currently a fake value v) is accessed, the result can be either undefined or a new fake value. If the operation returns a new fake value, the fake value's property prop refers to the new fake value.

The critical operations in forced execution are detailed as follows. These operations are related to well-known JavaScript language features.

Property read operations For an operation that attempts to read property p from a fake value v, if p is a string, there are the following cases:

(1) If the fake value v has been inferred to be a string, the concrete value of v is determined to be a randomly generated string s. The operation returns $s[p]$.
(2) If the fake value v has been inferred to be an array, the concrete value of v is determined to be an array a filled in with some randomly generated objects (or strings, numbers, etc.). The operation returns $a[p]$.
(3) If the fake value v has been inferred to be a number since the numbers in JavaScript are not extensive, the operation returns Number.prototype[p].
(4) If the type of fake value v is not determined, the operation can return undefined, or return a new fake value and set $v[p]$ to the new fake value.

However, when the property p is a well-known symbol [31], Diagnose considers three typical cases:

(1) Symbol.toStringTag [34], the operation can return strings that represent the built-in types, e.g., 'String', 'Number', 'Object', 'Function', 'Date', or a random string. If a string that represents a built-in type is returned, the type of the fake value is determined. If a random string is returned, the fake value is determined to be an object.
(2) Symbol.toPrimitive [33], a function is returned to create primitive values: if the input hint is 'number', the function returns a random number and determines the fake value v to be that random number. If the input hint is 'string', the function returns a random string, and v is determined to be the random string.
(3) Symbol.iterator [32], the fake value v is inferred to be an array, with a series of random values. The operation returns a generator function is returned to yield these random values one by one.

Property write operations For an operation that attempts to set property p to e on a fake value v, if o has been determined to be a number or a string, the operation makes no sense since strings and numbers in JavaScript are sealed. Otherwise, the property p is set to e.

***Typeof* operations** This operation corresponds to JavaScript typeof operator on a fake value v. It can return one of 'string', 'number', and 'object'. If the operation returns 'string', v is set to be a fake string. If the operation returns 'number', v is set to be a fake number.

Call operations This operation is recorded when a fake value v is invoked, hence v is determined to be a function, and a new fake value is returned. When v has been determined to be a string, a number, or an array, the operation throws an error since it cannot be invoked.

***Instanceof* operations** The operation corresponds to instanceof operator in JavaScript [30]. If the expression "v instanceof f" returns true, v is determined to inherit from f.prototype, otherwise v is not changed after the operation.

***GetOwnKeys* operations** The operation occurs when JavaScript built-in function Object.keys[5] is applied to a fake value v. The fake value v is determined to be an object, and Diagnose may choose to generate a series of fake key-value pairs to fill in the fake object. Then the operation returns the array of current keys.

The following example shows how Diagnose records the operations in forced execution.

[5]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/keys

```
 1 class Command {
 2   command(nameAndArgs, actionOptsOrExecDesc, execOpts) {
 3     let desc = actionOptsOrExecDesc;
 4     let opts = execOpts;
 5     if (typeof desc === 'object' && desc !== null) {
 6       opts = desc;
 7       desc = null;
 8     }
 9     opts = opts || {};
10     const [, name, args] = nameAndArgs.match(/([^ ]+) *(.*)/);
11     const cmd = this.createCommand(name);
12     if (desc) {
13       cmd.description(desc);
14       cmd._executableHandler = true;
15     }
16     if (opts.isDefault) this._defaultCommandName = cmd._name;
17     ...
18   }
19 }
```

Fig. 3. Method command in *commander*

Example 3. Consider the code from the method command in class Command[6] (shown in Figure 3), we describe how the property of input arguments are determined in the execution. Note that before command is called, the arguments and this are set to fake values.

(1) In Lines 3 and 4, the second and the third arguments are assigned to desc and opts, respectively.
(2) In Lines 5 to 8, there is a *typeof* operation on the fake value referenced by the local variable desc. In one execution, the operation returns 'object', therefore the fake value is determined to be an object rather than a string or number, and then the fake value is assigned to opts as the option object.
(3) In Line 9, the local variable opts is not changed since opts currently points to an object.
(4) In Line 10, the property match of the fake value referenced by nameAndArgs is called as a function, i.e., there is a *property read* operation on the object referenced by nameAndArgs, and a *call* operation on the object referenced by nameAndArgs.match. Hence the fake value should have match method, which accepts a regular expression and returns an array.
(5) In Line 11, the property createCommand of the fake object referenced by this is called with a fake value (referenced by name), hence the fake object referenced by this should have the property createCommand, which is a function.
(6) During the execution of createCommand, the property _registerCommand of this is accessed (not shown in Figure 3), and the referenced object is called. Therefore, the object referenced by this should have a property named _registerCommand, which is a function. In addition, the returned object of the function createCommand should contain the property copyInheriteSettings, which is a function.
(7) In Line 12, since desc is null now, the if branch is skipped.
(8) In Line 16, the property isDefault of opts is accessed, hence there is a *property read* operation on the object referenced by opt, and in this execution, the operation returns a new fake value, hence the fake object referenced by opts has the property isDefault.

To sum up, in this execution:

[6] https://github.com/tj/commander.js/blob/master/lib/command.js

```
{
  this:
  '{ createCommand: function() { return {
      copyInheritedSettings: function() { }
       } }, _registerCommand: function() {
      } }',
  arguments:
  [
    '{ match: function() { return [{}, {},
        {}] } }',
    '{ isDefault: {} }'
  ]
}
```

(a) Possible Input 1

```
{
  arguments: [
    '{ match: function() {
        return [{}, {}, {}] } }
        ',
    '\"fakestring\"',
    '{ isDefault: {} }'
  ]
}
```

(b) Possible Input 2 (this is omitted)

Fig. 4. Possible Input for the method command in class Command (the content of each field is the code to construct the corresponding object, which can be used to restore the corresponding object in the reconstruction phase)

(1) this should refer to an object having properties createCommand and _registerCommand.
(2) The first argument should have a match method, which returns an array containing three objects.
(3) The second argument is inferred to be an ordinary object that may contain isDefault property.
(4) The third argument can be any value, i.e., there can be only two arguments when the method command is called.

For this execution, we can gain some knowledge of the arguments. However, only some characteristics are not determined. Specifically, the first argument's property match is still a fake function, and the second argument's property isDefault is still a fake value. Therefore Diagnose replaces the fake values with randomly generated values. Diagnose then calls the method command with all arguments filled in with actual values and retains the argument sets that can obtain a non-error result. Diagnose also records the type returned from a function call. Finally, Diagnose obtains a series of input argument sets for a function to create *Call* edges (described in Section 4.2). During the execution, the built-in libraries related to I/O (e.g., fs, http, net) are mocked, like the previous work using forced execution [25].

As an example, Diagnose may create the input data shown in Figure 4a (represented in JSON format, this field and arguments field contain the content of the this and arguments used to call the corresponding function, respectively).

Since an operation on a fake value can have multiple effects, and a fake value might be implicitly coerced to a random primitive value, Diagnose can try to explore multiple execution paths. For example, if the *typeof* operation in Line 5 returns 'string', the second argument will be determined to be a string and the third argument to be an object having a property named isDefault. After the execution, Diagnose can create another set of input data, shown in Figure 4b (this is omitted since it is the same as this in Figure 4a).

4.2 Constructing Dynamic Object Relation Graph

In a CommonJS module, functions and other objects are usually organized in an exported object. Diagnose employs the dynamic object relation graph to maintain the structure of a JavaScript library and iteratively explore the JavaScript library (cf. insight 1). In this section, we detail the design of such structure and how to construct object relation graphs.

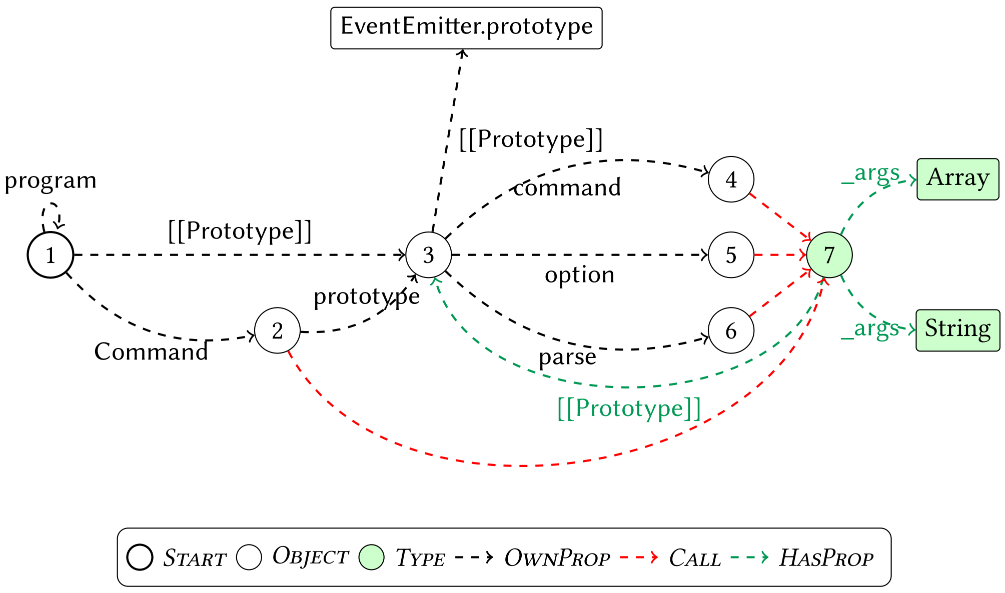


Fig. 5. Part of Dynamic Object Relation Graph for Commander (the input values on *Call* edges are omitted due to space limitation).

In a dynamic object relation graph, there are three types of nodes, i.e., *Start*, *Object* and *Type*, three types of edges, i.e., *Call*, *OwnProp* and *HasProp*. Specifically, the *Start* nodes and *Object* are associated with an object in the heap snapshot, and *OwnProp* and *HasProp* edges have a label. We detail each type of node and edge as follows.

- *Start* node: A *Start* node represents the loaded module in a dynamic object relation graph.
- *Object* node: An *Object* node represents an ordinary object in the heap snapshot after the library is loaded (including built-in objects like String and Object). The *Start* node is a special *Object* node.
- *Type* node: A *Type* node serves as the return type of functions in a library. It will be added during the forced execution-based type analysis process (in Section 4.1). The *Type* node can point to a series of Object nodes as its argument types and return type.
- *Call* edge: A *Call* edge connects an *Object* node and a *Type* node, which means that the *Object* node can be called as a function and can return an object whose type is represented by *Type* node. A *Call* edge is linked with a set of input arguments (including this) for a function call, which is described in Section 4.1.
- *OwnProp* edge: An *OwnProp* edge indicates an own member of an object. If the object represented by a node *A* has an *own property*,[7] referring to an object *B*, then there is a *OwnProp* edge from *A* to *B*. The own property can be a special property [[Prototype]], which refers to the prototype of the object.[8]
- *HasProp* edge: A *HasProp* edge links a *Type* node to another *Type* node which represents the type of a property, or *Object* node which represents the fixed value of a property.

[7]Own properties refer to those properties directly defined on an object, not inherited from the prototype.
[8]The property [[Prototype]] cannot be pragmatically used. See https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain.

The high-level process of object relation graph construction is as follows. Diagnose iteratively selects an object from the work list until the work list is empty or the max iteration time is reached. In each iteration, there are two critical processes:

- Diagnose explores more objects via property access, then adds them to the work list.
- If the object is a function, Diagnose does forced execution-based type analysis, and creates type nodes. More objects may be detected in this process and if they are not visited, they will be added to work list.

After the object is processed, it is set to be *visited*. We explain the two critical processes as follows.

The process of exploring objects via property access. The procedure is roughly a breadth-first search starting from an unvisited node in the work list. Diagnose finds new nodes via *own properties*. Diagnose then creates nodes for all reachable and unvisited objects (except built-in objects, e.g., Object) and adds them to the work list (if unvisited) for subsequent processes. We use the dynamic object relation graph in Figure 5 to illustrate the process:

(1) Node 1 represents the starting object, i.e., the imported library. Assuming that the loaded library is assigned to the variable lib, i.e., **var** lib = require('commander'). There is a self loop on Node 1, since lib and lib.program refer to the same object.
(2) Nodes 2 represent the object referenced by own properties Command of the object represented by Node 1, respectively. Node 2 can be accessed via lib.Command.
(3) The object represented by Node 2 is a function and can be called as a constructor, hence it has an *own* property named prototype, which refers to the object represented by Node 3. Hence, Node 3 can be accessed via lib.Command.prototype.
(4) The object represented by Node 3 has a number of properties representing a function, e.g., command, option and parse, represented by Node 4, 5 and 6 respectively.
(5) The prototype of the object represented by Node 3 is EventEmitter.prototype [36], which is a JavaScript built-in object. Hence Node 3's *OwnProp* edge named [[Prototype]] points to EventEmitter.prototype.
(6) Regarding the object represented by Node 1, its prototype is the object represented by Node 3, hence there is an *OwnProp* edge named [[Prototype]] from Node 1 to 3.

The process of forced execution based analysis. In the process, Diagnose performs a type analysis for the objects that are functions and not visited. Diagnose creates *Call* edges and *Type* nodes according to the analysis results. In type analysis, new objects may be detected, e.g., one property of the returned object is a new prototype object in the return type. In this case, Diagnose will create new *Object* nodes for these objects.

Example 4. In Figure 5, the object represented by Node 2 and Nodes 4 to 6 are functions:

(1) The returns objects of the function represented by Node 2 have a fixed set of properties and a common prototype. Hence Diagnose creates a *Type* node (Node 7) and a *Call* edge from Node 2 to 7. Since the common prototype can be represented by Node 3, Diagnose creates a *HasProp* edge named [[Prototype]] from Node 7 to 3. Additionally, Diagnose creates a series of *HasProp* edges starting from Node 7 (we here only show two edges, i.e., _args and _name).
(2) The return types of functions (represented by Node 4, 5 and 6) can be represented by Node 7, i.e., having a common prototype and a fixed set of properties. Hence Diagnose creates *Call* edges from Node 4, 5, 6 to Node 7, respectively.

4.3 Graph Refinement

The major purpose of refinement is to make the test input values of the *Call* edges reflect the practical usage. Diagnose uses the following refinement methods.

```
{
  this: '__node__[1].object.apply(
      __node__[1].calledges[0].
      this, __node__[1].calledges
      [0].arguments)',
  arguments: [
    '{ match: function() { return
        [{}, {}, {}] } }',
    '{ isDefault: {} }'
  ]
}
```

(a) Refine Input 1

```
{
  arguments: [
    '"fakestring"',
    '{ isDefault: {} }'
  ]
}
```

(b) Refined Input 2 (this is the same as Input 1)

Fig. 6. Refined Input Data for command

Refine the input values on* Call *edges. If the type of an argument (or this) might be created by calling another function in the dynamic object relation graph or a native constructor, i.e., the return type of the function is a *subtype* of the inferred. This can avoid direct use of internal properties and identify native types (insights 2 and 3).

We say that the type t a subtype of t' (written $t <: t'$) if $\text{PropertySet}(t') \subseteq \text{PropertySet}(t)$ and for each $p \in \text{PropertySet}(t')$, $t[p] <: t'[p]$. This ensures that t is compatible with t'. For example, as shown in Figure 4, the first input argument is an object having a function property named match, which returns an array. Note that JavaScript native strings have a match method (inherited from String.prototype), which returns an array (with undetermined length) or null. Hence, the type of String.prototype.match[9] is a subtype of the inferred since the latter returns an array with three elements, while the former returns an arbitrary array or null. And further, the native String type is a subtype of the type of the inferred argument.

Example 5. Continuing Example 3, the object referenced by this for the function represented by Node 4 in Figure 5 (i.e., the method command shown in Figure 3) should have properties createCommand and _registerCommand, which are both functions. However, the return objects of Command(...) share a common prototype object, which owns the function properties named createCommand and _registerCommand, respectively. Hence, this object can be replaced with the objects created by Command(...), if command can successfully run after the input object for this is replaced. A possible set of refined input values on the *Call* edge is shown in Figure 6a, where the code for this field is to retrieve the object created by the function represented by Node 2 with the input values on its first *Call* edge[10].

Additionally, the first argument of command is an object with the function property named match. The function returns an array. The strings in JavaScript also have a function property named match (inherited from String.prototype), which returns an arbitrary array or null. Hence, the argument can be replaced by a random string, as shown in Figure 6b.

Prune the graph. Diagnose employs a simple process to remove the following edges and nodes since they possibly represents the objects and properties for internal use:

(1) If a *HasProp* edge points to a *Type* node, the edge should be removed.
(2) If an *Object* node has no outer edges, the node should be removed.

[9]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/match
[10]Here the identifier __node__ is a special token to access the nodes in the dynamic object relation graph.

(3) If a node in the dynamic object relation graph is not connected to any other node, i.e., a detached node, it should be removed.

For example, in Figure 5, the properties _args and _name of Node 7 should be removed. They are probably only for internal usage, hence should not be considered as the type-related information of the library.

Supplement edges and nodes based on example client code. We adopt the example code of earlier versions of a JavaScript library to refine the dynamic object relation graph. By running the example code with the loaded library wrapped, Diagnose can record the input and return values of function objects. If new objects are discovered in the heap snapshot or new types are produced, Diagnose will correspondingly create new nodes and add them to the work list for subsequent analysis.

Example 6. The example code can access the methods in *commander* as follows:

```
1 const { Command } = require('commander');
2 const program = new Command();
3 program
4   .command('build')
5   .description('build web site for deployment')
6   .action(() => {
7     console.log('build');
8   });
```

Since Diagnose has built a dynamic object relation graph for *commander*, when executing the code above, it can force the return object of require('commander') to be the *Start* node of the dynamic object relation graph (shown in Figure 5). Then Command will refer to the wrapped Node 2 in Figure 5. when **new** Command() is executed, there is no extra argument supplied, and Diagnose will record it to create a new *Call* edge. In Line 4, the command property refers to the wrapped command method, which can record the input values when calling this method (here the input value is a single string 'build'). Hence, Diagnose can create another *Call* edges with the input value 'build', which has different arguments compared to those in Figure 4 and 6. Similarly, Diagnose can process the method description and action (in Line 5 and 6, respectively).

4.4 Breaking Change Detection by Graph Reconstruction

Diagnose generates a full object relation graph for the older revision of a library. Diagnose will try to check whether the dynamic object relation graph can be rebuilt on the newer version of the library. The process is similar to graph construction explained in Section 4.2, however, Diagnose only explores the objects and creates *Type* nodes follow the edges in the original object relation graph. We use the dynamic object relation graph in Figure 5 as an example (Node i' means the node in the reconstructed dynamic object relation graph corresponding to Node i in the original graph):

(1) Diagnose first loads the new version of the library *commander* and creates a *Start* node (Node $1'$) for it, corresponding to Node 1 in Figure 5.
(2) Since there are two *OwnProp* edges starting from Node 1, named program and Command respectively, Diagnose then checks the existence of the two properties on the *Start* node. If not, there should be a breaking change. Otherwise, Diagnose creates two *OwnProp* edges starting from the *Start* node (i.e., Node $1'$) and an *Object* node (i.e., Node $2'$).
(3) Node 2 has an own property prototype, referring to Node 3 in Figure 5. Then Diagnose checks whether the object represented by Node $2'$ has an *own* property named prototype. If

Table 2. Overall Effectiveness Comparison between Different Approaches

Approach	Detected	Missed	False Positive
Dont-break	9	167	1
NoRegrets+	19	157	42
Tsinfer-node	14	162	46
Diagnose	106	70	9

not, there should be a breaking change. Otherwise, Diagnose creates Node 3′, as well as an *OwnProp* edge from Node 2′ to 3′.

(4) The prototype of the object represented by Node 1 is the object represented by Node 3, hence Diagnose checks whether the prototype of the object represented by Node 1′ is actually the object represented by Node 3′. If not, there should be a breaking change. Otherwise, Diagnose creates an *OwnProp* edge named [[Prototype]] from Node 1′ to 3′.
(5) Similarly, Diagnose checks Node 4, 5, 6 and EventEmitter.prototype.
(6) In Figure 5, there is a *Call* edge from Node 2 to 7. Diagnose hence calls the function object represented by Node 2′ with the input value sets on the *Call* edge and checks whether the return value conforms to the type represented by Node 7. Since the properties _args and _name are removed in the refinement process, Diagnose only checks [[Prototype]], i.e., whether the prototype of the return value is the object represented by Node 3′. If any errors are thrown in the process or the type check fails, there should be a breaking change.
(7) Similarly, Diagnose checks the *Call* edges from Node 4, 5, 6 to Node 7.

4.5 Evaluation

In this section, we present the experimental results to show the effectiveness and efficiency of our proposed approach. We focus on three research questions:

RQ1: How many breaking changes can Diagnose detect in popular JavaScript libraries?
RQ2: Can Diagnose produce fewer false positives?
RQ3: How efficient is Diagnose and whether it is suitable for practical use?

During evaluation, the benchmark libraries include the same set of JavaScript libraries used in Section 3 that can be packaged into CommonJS modules. Other experiment settings are similar to those used in Section 3.1. Due to space limitation, we only present the overall performance statistics of Diagnose and other approaches in Table 2 and put our selected libraries and the experimental results in our replication package [23].

RQ1: The number of detected breaking changes. Diagnose can detect 106 breaking changes (60.2%). Notably, for larger libraries (typically containing plenty of JavaScript source code files and having many version releases), such as *joi* and *mongoose*, Diagnose can detect and much more breaking changes. This mainly benefits from the exploration of callable objects and execution paths in the JavaScript library (as is discussed in Section 3, other approaches does not explore APIs in generating API models). Furthermore, since Diangose can construct more possible inputs for exposed APIs, it can detect more breaking changes.

The following example illustrates how Diagnose can detect more breaking changes compared to NoRegrets+.

Example 7. In *immutable* 4.0.0, the behavior of the function isImmutable is changed: it returns true for all immutable collections, even within withMutation calls[11] (in withMutation calls, the

[11]https://github.com/immutable-js/immutable-js/pull/1374

collections will be temporarily transformed to be mutable by calling asMutable). For example, the function call isImmutable(Map().asMutable()) will return true in version 4.0.0. In the code change of version 4.0.0, the function isImmutable no longer accesses the property __ownerID of the argument. It can indicate that there might be a breaking change. However, NoRegrets+ only regards additional property read operation as a breaking change. Hence, NoRegrets+ does not report the breaking change.

By contrast, Diagnose does not detect the breaking change from the aspect of property access, since the changes of internal property access do not definitely means a breaking change (especially in refactoring, the internal properties can change a lot). It first analyzes the input type of the function isImmutable via forced execution: the input argument might have some internal properties, e.g., @@__IMMUTABLE_ITERABLE__@@, __ownerID, etc., and construct the possible input arguments. Also, the return values of Map.prototype.asMutable contain such properties, hence in the process of graph refinement, Diagnose can replace the arguments with the objects created via Map().asMutable(). After the dynamic object relation graph is constructed, Diagnose can try to construct the graph in the newer version of *immutable* and call the function isImmutable with the arguments created via Map().asMutable(), then the function call returns true rather than false in the previous version, thereby the breaking change is detected.

RQ2: The number of false positives. Diagnose only reports 9 false positives while NoRegrets+ and Tsinfer-node reports 42 and 46 respectively. One reason is that Diagnose does not regard changes in internal properties as breaking changes: in some major version updates such as *mongoose* 5.0.0 to 6.0.0, the internal structure of some types may be largely refactored, Diagnose can ignore them by pruning the dynamic object relation graph, described in Section 4.3. The following example can illustrate how Diagnose avoids such false positives.

Example 8. This example is provided by Møller et al. [29]. In the library *joi*, the uri method of the return type of joi.string() can be used as follows:

```
var joi = require('joi');
var v = joi.string().uri({scheme: 'http'});
var result = v.validate('http://example.com');
// result.error is currently null
```

If no error occurs in the execution of v.validate(...), the property error of result is null. In version 13.5.0, a new option named allowQuerySquareBrackets is introduced. Hence the method uri will read the property. However, for simple URIs, the result will not change. Moreover, the option defaults to false and the method preserves the old behavior when the option is not set or is false. Although it is only a benign change, not breaking change, NoRegrets+ still reports it since in the execution of uri, the new option allowQuerySquareBrackets is read. As for Diagnose, it will only report a breaking change if the type related information of return objects is changed (e.g., the property error is no longer null). However, in this example, the return type of the v.validate(...) call has no change after the version update, hence Diagnose avoids such a false positive.

RQ3: Efficiency of Diagnose. In our evaluation, we consider two measures, i.e., the mean time of constructing the dynamic object relation graph and the mean time of reconstructing the dynamic object relation graph for an updated version. We implement Diagnose in pure JavaScript and evaluate Diagnose on a cloud computer with 2-core CPU and 8 GB memory. Overall, the mean time for constructing an dynamic object relation graph for a specific version of the libraries is 1.4 minutes. For very small libraries like *camelcase*, *has-flag* (with only a single JavaScript source code file), Diagnose can construct the dynamic object relation graph within 15 seconds. Even for the largest library *mongoose*, Diagnose can finish the construction process within 8 minutes and the reconstruction process for the subsequent versions within 3 minutes (with max iteration of 100 and max generated test input of 100). Hence, DIAGNOSE is suitable in practical use.

5 Discussion

5.1 Comparison with NoREGRETS+

DIAGNOSE and NoREGRETS+ both generate *models* for a Node.js library and employ model-based testing for the updated libraries. However, they differ in the following aspects.

Most importantly, NoREGRETS+ only supports basic built-in types, including number, string, object, function, and other types in the JavaScript standard library (e.g., date, Map, EventEmitter). For object type, NoREGRETS+ only supports the properties that are accessed in the client code. By contrast, DIAGNOSE support not only the built-in types, but also the types in the JavaScript libraries by relating the return types and input types in JavaScript libraries, e.g., inferring that the returned type of a function can be used as the input as another function, and refine the input values on the *CALL* edges.

Secondly, the model generated by NoREGRETS+ is subject to a test suite, i.e., can only be used with the help of the test suite. That is not satisfactory when it is not easy to find test cases for a JavaScript library. By contrast, DIAGNOSE can automatically infer types of exported functions in JavaScript libraries via forced execution-based type analysis (detailed in Section 4.1), and generate concrete and runnable test cases for the exported functions.

Last, NoREGRETS+ considers multiple calls of an exported function in client code. Typically, when a function named f is called twice in client code, possibly with different types of input and returned values, they are both recorded. However, DIAGNOSE achieves this in a different way. DIAGNOSE can explore the possible types of this, input and return values via forced execution-based type analysis and then generate input value sets that can make the functions successfully run (detailed in Section 4.1). Hence, it is not necessary for DIAGNOSE to analyze client code with multiple API calls of a function.

5.2 Limitation of DIAGNOSE

While DIAGNOSE can detect more breaking changes compared to prior tools, it cannot handle those breaking changes only related to program behavior. For example, in *mongoose* 6.0.0, the developers made the method Document.prototype.$set set keys in the order they were defined in the schema, not in the user-specified object[12]. As a result, the returned objects are not ordered as expected. However, DIAGNOSE only consider type changes as breaking changes.

Nevertheless, in practice, determining whether a behavior change is breaking is often difficult, even for developers of JavaScript libraries. For different downstream users, the impacts of a behavior change may vary largely, and some code changes are identified as breaking changes by the upstream developers, but impact little on downstream (e.g., the example above). Hence, ignoring these changes is reasonable, and is possible to avoid many false positives.

6 Threats to Validity

External Threats. In our study, we just evaluate existing tools and our approach on 40 well-known libraries selected from the most popular JavaScript libraries in the NPM registry. The results might not fully reflect the breaking changes in the libraries that are rarely used. However, the breaking changes in popular libraries are much more critical, and unpopular libraries often do not have many client test cases to uncover the code changes in version updates. Therefore, we believe that considering breaking changes in the libraries with the most dependent projects is reasonable.

[12]https://github.com/automattic/mongoose/issues/4665

Internal Threats. In the evaluation process of existing tools and analyze breaking changes that are documented by developers, we might incorrectly understand the developers' intention (e.g., considering a normal bug fix as a breaking change) and wrongly identify the actual reasons why these tools cannot detect some breaking changes or produce false positive. To mitigate this, we carefully check the running results and try our best to understand how the tools produce such results.

Construct Threats. The construct threats mainly relate to the quality of our benchmarks, since determining breaking changes is really quite challenging. Møller et al. [29] pointed out that API specification should be constructed as a requirement of semantic versioning. However, it is often not practical. Typically, the correct API usage might not be present in the API documentation. Hence if the client code incorrectly utilizes the API and finds the behavior changes after library updates, we cannot definitely say that there is a breaking change. For example, after a library update, one API's return objects do not have a property (which is for internal use only). Besides, developers might overestimate the effects of code changes, e.g., identifying a subtle change as a breaking change, or overlook some breaking changes that will actually break the client code. However, our selected libraries are the most popular ones from the NPM registry, hence they are more likely to be well-maintained, and the breaking changes are correctly documented, and developers are less probable to overestimate the impacts of breaking changes.

7 Related Work

Program Analysis for JavaScript. Program analysis can be utilized to determine a lot of syntactic and runtime behavior for JavaScript programs, and can be further used to detect potential bugs and breaking API changes. Some works proposed static analysis approaches from many aspects. For example, Madsen et al. [26] built event-based call graphs to detect event-related bugs by enhancing the static analysis framework JASI [19] and TAJS [17]. Other works [17, 37, 44] studied static analysis for JavaScript in the HTML DOM environment. Furthermore, in addressing the limitations of static analysis approaches for JavaScript, Chakraborty et al. presented a technique to supplement missing edges in the JavaScript call graph [4] and highlighted that dynamic property access is a primary factor contributing to low recall in previous static analysis frameworks, mainly missing some function calls. Besides, some open-source tools like ESLint [13] and JSHint [18] support rule-based static analysis for JavaScript projects. They can be used to improve code quality, e.g., making code follow JavaScript programming idioms. While static analysis techniques are effective in many scenarios, they still suffer from some problems, typically the limitation in the approximation of the runtime behavior. Hence, dynamic analysis can be the tendency of JavaScript program analysis. For example, Pradel et al. proposed TypeDevil [39] that builds runtime type relation graphs to detect type inconsistency in JavaScript programs, Gong et al. proposed JITProf [15] to analyze runtime performance problems.

Breaking Change Analysis in JavaScript. To the best of our knowledge, there are only few works aimed at breaking change analysis in JavaScript. There is an empirical study [22] that investigated the breaking changes from the developers' aspects, i.e., when, how, and why developers perform a breaking change. Specifically, they investigated the source code-level features of JavaScript breaking changes (including syntactic breaking changes and behavioral breaking changes). However, the scenario in our paper is different. We focus on client users' aspects. Our investigation is based on the version level: when client users (a.k.a., downstream developers) have an older version of a library, now they need to update to a newer version. We do not focus on how developers change the source code. Additionally, the libraries that they used are collected from the Libraries.io dataset [20], which was released in 2020, and many of them are not popular nowadays. TAPIR [28] is a semi-automatic tool for detecting the client code that is affected by breaking changes. By contrast, our approach Diagnose is aimed at detecting breaking changes. Nevertheless, in the future, it is possible to extend Diagnose to generate such statements of the formal language to describe the detected breaking changes.

Breaking Change Detection and Analysis for Other Languages. Several studies focus on detecting and analyzing breaking changes for other programming languages, especially Java and Python, and their techniques might be learned to boost breaking change detection for JavaScript. Regarding syntactic breaking changes, APIDiff proposed by Brito et al. [3] can detect syntax-related breaking changes in Maven projects, such as method removal and visibility loss by reusing the refactoring detection tool RefDiff [43]. Some open-source tools can also check Java syntactic breaking changes, such as Clirr [5] and RevAPI [41]. For Python language, Du et al. [12] proposed AexPy that can detect similar types of breaking changes like module removal and addition of required parameters, which extends the existing tool PyCompat [47] and Pidiff [38]. As for non-syntactic breaking changes, Zhang et al. proposed Sembid [46] to detect behavioral breaking changes by measuring the semantic difference of call graphs between old and new programs. However, Sembid cannot detect subtle breaking changes and may report large refactorings as breaking changes.

8 Conclusion and Future Work

In this work, to illustratively understand the effectiveness of current approaches for JavaScript breaking change detection, we preliminary evaluate the approaches on 40 popular JavaScript libraries sampled from the NPM registry. We find that the existing approaches miss a lot of breaking change and might report false positives, and discuss the reasons. We further provide insights for further improvements, and based on them, we propose a new approach named Diagnose that constructs dynamic object relation graphs for JavaScript libraries with forced execution-based type analysis. Diagnose then checks whether the graphs can be reconstructed in the updated versions of the libraries. Compared to existing tools, Diagnose can detect significantly more breaking changes and produce much fewer false alarms. In the future, we plan to extend Diagnose to support more JavaScript features and apply it to more JavaScript-related tasks, such as vulnerability detection.

Data Availability

We provide the replication package and supplementary material [23].

Acknowledgments

We sincerely thank the anonymous reviewers for their comments and insightful feedback. This research (project) is supported by the National Science Foundation of China (No.62372398 and No.72342025), the Zhejiang Pioneer (Jianbing) Project (2025C01198(SD2)), and funded by ZJU-China Unicom Digital Security Joint Laboratory. The research (project) is also supported by the National Research Foundation, Singapore, under its Investigatorship Grant (NRF-NRFI08-2022-0002). Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

References

[1] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2016. How to break an API: cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 109–120.

[2] Chris Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. 2021. When and how to make breaking changes: Policies and practices in 18 open source software ecosystems. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 4 (2021), 1–56.

[3] Aline Brito, Laerte Xavier, Andre Hora, and Marco Tulio Valente. 2018. APIDiff: Detecting API breaking changes. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 507–511.

[4] Madhurima Chakraborty, Renzo Olivares, Manu Sridharan, and Behnaz Hassanshahi. 2022. Automatic root cause quantification for missing edges in javascript call graphs. In *36th European Conference on Object-Oriented Programming (ECOOP 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[5] Clirr. 2024. Clirr. https://clirr.sourceforge.net.

[6] Fernando Cristiani and Peter Thiemann. 2021. Generation of typescript declaration files from javascript code. In *Proceedings of the 18th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes*. 97–112.

[7] Daniela S Cruzes and Tore Dyba. 2011. Recommended steps for thematic synthesis in software engineering. In *2011 international symposium on empirical software engineering and measurement*. IEEE, 275–284.

[8] Alexandre Decan and Tom Mens. 2019. What do package dependencies tell us about semantic versioning? *IEEE Transactions on Software Engineering* 47, 6 (2019), 1226–1240.

[9] DefinitelyTyped. 2025. DefinitelyTyped. https://github.com/definitelytyped/definitelytyped.

[10] Dont-break. 2024. Dont-break. https://github.com/bahmutov/dont-break.

[11] Dts-gen. 2025. Dts-gen. https://www.npmjs.com/package/dts-gen.

[12] Xingliang Du and Jun Ma. 2022. AexPy: Detecting API Breaking Changes in Python Packages. In *2022 IEEE 33rd International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 470–481.

[13] ESLint. 2025. ESLint. https://eslint.org.

[14] Asger Feldthaus and Anders Møller. 2014. Checking correctness of TypeScript interfaces for JavaScript libraries. In *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications*. 1–16.

[15] Liang Gong, Michael Pradel, and Koushik Sen. 2015. Jitprof: Pinpointing jit-unfriendly javascript code. In *Proceedings of the 2015 10th joint meeting on foundations of software engineering*. 357–368.

[16] Xunchao Hu, Yao Cheng, Yue Duan, Andrew Henderson, and Heng Yin. 2018. Jsforce: A forced execution engine for malicious javascript detection. In *Security and Privacy in Communication Networks: 13th International Conference, SecureComm 2017, Niagara Falls, ON, Canada, October 22–25, 2017, Proceedings 13*. Springer, 704–720.

[17] Simon Holm Jensen, Magnus Madsen, and Anders Møller. 2011. Modeling the HTML DOM and browser API in static analysis of JavaScript web applications. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*. 59–69.

[18] JSHint. 2024. JSHint. https://jshint.com.

[19] Vineeth Kashyap, Kyle Dewey, Ethan A Kuefner, John Wagner, Kevin Gibbons, John Sarracino, Ben Wiedermann, and Ben Hardekopf. 2014. JSAI: A static analysis platform for JavaScript. In *Proceedings of the 22nd ACM SIGSOFT international symposium on Foundations of Software Engineering*. 121–132.

[20] Jeremy Katz. 2020. *Libraries.io Open Source Repository and Dependency Metadata*. https://doi.org/10.5281/zenodo.3626071

[21] Kyungtae Kim, I Luk Kim, Chung Hwan Kim, Yonghwi Kwon, Yunhui Zheng, Xiangyu Zhang, and Dongyan Xu. 2017. J-force: Forced execution on javascript. In *Proceedings of the 26th international conference on World Wide Web*. 897–906.

[22] Dezhen Kong, Jiakun Liu, Lingfeng Bao, and David Lo. 2024. Towards Better Comprehension of Breaking Changes in the NPM Ecosystem. *ACM Transactions on Software Engineering and Methodology* (2024).

[23] Dezhen Kong, Jiakun Liu, Chao Ni, David Lo, and Lingfeng Bao. 2025. Diagnose Replication Package. https://github.com/cstimkong/diagnose.

[24] Erik Krogh Kristensen and Anders Møller. 2017. Inference and evolution of typescript declaration files. In *Fundamental Approaches to Software Engineering: 20th International Conference, FASE 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings 20*. Springer, 99–115.

[25] Mathias Rud Laursen, Wenyuan Xu, and Anders Møller. 2024. Reducing static analysis unsoundness with approximate interpretation. *Proceedings of the ACM on Programming Languages* 8, PLDI (2024), 1165–1188.

[26] Magnus Madsen, Frank Tip, and Ondřej Lhoták. 2015. Static analysis of event-driven Node. js JavaScript applications. *ACM SIGPLAN Notices* 50, 10 (2015), 505–519.

[27] 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)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[28] Anders Møller, Benjamin Barslev Nielsen, and Martin Toldam Torp. 2020. Detecting locations in JavaScript programs affected by breaking library changes. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–25.

[29] Anders Møller and Martin Toldam Torp. 2019. Model-based testing of breaking changes in Node. js libraries. In *Proceedings of the 2019 27th ACM joint meeting on european software engineering conference and symposium on the foundations of software engineering*. 409–419.

[30] Mozilla. 2025. JavaScript Instanceof Operator. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/instanceof.

[31] Mozilla. 2025. JavaScript Symbol. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol.

[32] Mozilla. 2025. JavaScript Symbol.iterator. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol/iterator.

[33] Mozilla. 2025. JavaScript Symbol.toPrimitive. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol/toPrimitive.

[34] Mozilla. 2025. JavaScript Symbol.toStringTag. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Symbol/toStringTag.

[35] Benjamin Barslev Nielsen, Martin Toldam Torp, and Anders Møller. 2021. Modular call graph construction for security scanning of Node. js applications. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 29–41.

[36] NodeJS. 2025. Node.js Event. https://nodejs.org/api/events.html.

[37] Changhee Park, Sooncheol Won, Joonho Jin, and Sukyoung Ryu. 2015. Static Analysis of JavaScript Web Applications in the Wild via Practical DOM Modeling (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 552–562. https://doi.org/10.1109/ASE.2015.27

[38] Pidiff. 2024. Pidiff. https://github.com/rohanpm/pidiff.

[39] Michael Pradel, Parker Schuh, and Koushik Sen. 2015. TypeDevil: Dynamic type inconsistency analysis for JavaScript. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 314–324.

[40] Steven Raemaekers, Arie van Deursen, and Joost Visser. 2017. Semantic versioning and impact of breaking changes in the Maven repository. *Journal of Systems and Software* 129 (2017), 140–158.

[41] RevAPI. 2024. RevAPI. https://revapi.org.

[42] SemVer. 2025. Semantic Versioning. http://semver.org.

[43] Danilo Silva and Marco Tulio Valente. 2017. RefDiff: Detecting refactorings in version histories. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE, 269–279.

[44] Chungha Sung, Markus Kusano, Nishant Sinha, and Chao Wang. 2016. Static DOM event dependency analysis for testing web applications. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 447–459.

[45] Daniel Venturini, Filipe Roseiro Cogo, Ivanilton Polato, Marco A Gerosa, and Igor Scaliante Wiese. 2023. I depended on you and you broke me: An empirical study of manifesting breaking changes in client packages. *ACM Transactions on Software Engineering and Methodology* 32, 4 (2023), 1–26.

[46] Lyuye Zhang, Chengwei Liu, Zhengzi Xu, Sen Chen, Lingling Fan, Bihuan Chen, and Yang Liu. 2022. Has my release disobeyed semantic versioning? Static detection based on semantic differencing. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*. 1–12.

[47] Zhaoxu Zhang, Hengcheng Zhu, Ming Wen, Yida Tao, Yepang Liu, and Yingfei Xiong. 2020. How do python framework apis evolve? an exploratory study. In *2020 ieee 27th international conference on software analysis, evolution and reengineering (saner)*. IEEE, 81–92.

Received 2024-10-31; accepted 2025-03-31; revised 2024-10-31; accepted 2025-03-31