Detecting Breaking Changes in JavaScript APIs

Master's Thesis

Michel Kraaijeveld

Detecting Breaking Changes in JavaScript APIs

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Michel Kraaijeveld
born in Werkendam, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

©2017 Michel Kraaijeveld. All rights reserved.

Detecting Breaking Changes in JavaScript APIs

Author: Michel Kraaijeveld
Student id: 4244311
Email: J.C.M.Kraaijeveld@student.tudelft.nl

Abstract

The goal of this thesis is to explore the current possibilities for detecting breaking changes in JavaScript. For this, we propose an approach and show its accuracy by constructing a tool and evaluating it.

The evaluation is carried out on 3 chosen JavaScript projects and a total of 3000 consumer packages. For each of the projects, we compute the precision and recall rates. Furthermore, an empirical study is carried out on the 3000 consumer packages to see the effects of breaking changes on developers. The results show that we are able to detect between 43% and 80% of breaking changes. The outcome of the empirical study suggests that breaking changes appear quite often between versions, and even in versions that should not contain them according to the rules for semantic versioning. Additionally, we show the current limitations of our approach and how they can be improved upon in future research.

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. M. Aniche, Faculty EEMCS, TU Delft
Committee Members:	Dr. A. Bacchelli, Faculty EEMCS, TU Delft
	Dr. C. Hauff, Faculty EEMCS, TU Delft

Contents

Contents iii

List of Figures v

1 Introduction 1

2 Background on the JavaScript Language 3
- 2.1 The JavaScript Language . . . 3
- 2.2 Functions & Parameters . . . 3
- 2.3 Modularization with Node.js . . . 5

3 An Approach to Detect Breaking Changes in JS Projects 7
- 3.1 Project Parsing . . . 8
- 3.2 Function Extraction . . . 10
- 3.3 Breaking Changes List Creation . . . 11
- 3.4 Consumer Package Parsing . . . 12
- 3.5 Function Detection . . . 13
- 3.6 Breaking Changes List Generation . . . 14

4 Methodology 15
- 4.1 Project Selection . . . 16
- 4.2 Creating the Oracle . . . 17
- 4.3 Data Analysis . . . 19

5 Results 21
- 5.1 RQ1: How accurate is our approach in detecting breaking changes? . . . 21
- 5.2 RQ2: In what cases does the tool produce incorrect results? . . . 23
- 5.3 RQ3: How often are projects affected by breaking changes? . . . 24
- 5.4 RQ4: To what extent do breaking changes affect consumer packages? . . . 26

6 Discussion of our Findings 29

6.1 Approach Accuracy 29
6.2 Performance of the Tool 30
6.3 Threats to Validity 32
6.4 Future Work 32

7 Related work **35**
7.1 Existing Tools 35
7.2 Change Detection 35
7.3 Dependencies in Software Systems 36
7.4 Breaking Changes in Software Systems 37

8 Conclusion **39**

Bibliography **41**

List of Figures

3.1 Approach Overview 8

4.1 Creating a consumer package oracle 20

5.1 Version distribution for consumer packages 27

Chapter 1

Introduction

Breaking changes are a well known phenomenon in any programming language. A famous example in JavaScript, where a single developer ”broke the internet”[7], is the left-pad incident. A developer unpublished multiple of his packages, of which one called Left-pad was depended upon by thousands of other projects. As a result, those projects did no longer work and since those projects were also depended upon by others, it set off a chain reaction. This shows how severe a breaking change can turn out to be. For typed languages, such as Java and .NET, there are existing tools that can detect breaking changes (e.g. [1, 5, 8, 9]. However, this is not the case for a dynamic language like JavaScript.

Therefore we present in this thesis an approach to detect breaking changes in JavaScript. Our focus is on the detection of breaking changes in project APIs, that are used by consumer packages. We limit ourselves to changes in function names and parameter amounts, discarding changes in types due to the dynamic nature of JavaScript. Although there are frameworks available that add types to JavaScript, those did not provide us with enough information, which we further explain in Section 3.

Furthermore, we implement our approach in the form of a tool. It makes use of the project's source code files, transforms them to an AST and detects relevant constructs to extract functions from. We also consider the possible use of React in server-side packages, as it includes additional syntax that not every parser can work with when generating an AST. We found it important to ensure that our tool does not fail when such projects are encountered.

An issue with the approach is that it will also contain private functions, while we are only interested in the public ones. To counter this, we make use of 3000 consumer packages to filter out private functions, which has two advantages. The first being that we will end up with breaking changes files for each of the consumers, for which we can check how well the tool performs by means of an empirical study. Secondly, by combining the results of the consumer packages, we are only left with the public functions, under the assumptions that no private functions are used by consumers.

From the results it follows that we are able to detect between 43% and 80% of breaking changes that are available in a given project, in case of individual packages this is between 28% and 77%. Furthermore, the empirical research indicate that only between 7.6% and 32% of consumer packages make use of the most recent version of a project. Additionally,

we found that 9.8% to 25.8% of consumers suffer from breaking changes if they decide to update to the latest version.

The thesis contains related work on change detection, dependencies and breaking changes. Although most related work focuses on typed languages such as Java and the corresponding Maven repository, they do bring up possible extensions to our work. Additionally, multiple studies report on dependencies being outdated[12, 14], with a possible reason being breaking changes. Also, it seems that in case of Java, breaking changes are equally likely to be introduced in minor and major version[19]. Our empirical findings also show breaking changes being introduced in minor versions, which could lead to developers unexpectedly breaking their code.

We provide multiple contributions with our work. First, we described an approach for detecting breaking changes in consumer packages and to generate a list of public breaking changes from them. Furthermore, the provide an implementation of the approach by creating a tool, which will be made available on GitHub[1]. Finally, we have conducted an empirical study on a total of 3000 consumer packages to evaluate our approach and implementation.

The remainder of the thesis is structured as follows. Chapter 2 introduces background information on JavaScript. Chapter 3 details our approach on detecting breaking changes, while Chapter 4 presents our research questions and our methodology, including the creation of our oracle and data analysis. The results are listed in Chapter 5, followed by a discussion of our findings and possible future work in Chapter 6. Chapter 7 contains related work in terms of change detection, dependencies and breaking changes. In Chapter 8 we present the conclusion of our thesis.

[1] https://github.com/WhatTheEffort

Chapter 2

Background on the JavaScript Language

Throughout this research, our focus lies on breaking changes in JavaScript. Therefore, we first introduce the JavaScript programming language and how functions in JavaScript can be defined in Sections 2.1 and 2.2. We also quickly touch the Node.js environment in Section 2.3, as it contains specifics that we rely on in our thesis.

2.1 The JavaScript Language

JavaScript, often abbreviated to JS, is a versatile programming language, primarily known as a front-end language for websites. However, it can also be used for back end applications or even as framework[1] to create apps for mobile devices. From a recent Stack Overflow survey[2] it follows that JavaScript is a very popular programming language, as it ranks the number one spot for both front-end and back-end development.

JavaScript is a dynamic language, which means that certain behaviour is determined at run-time rather than during compilation. To give an example of dynamic behaviour: types are only determined at run-time, so no type information will be known in the static code. Apart from this, JavaScript is also a prototype-based languages, which means that there are no actual classes. Instead, objects can inherit certain functionality and data from other objects.

2.2 Functions & Parameters

When a function changes, it can introduce a breaking change. To detect those breaking changes, it is important to understand the valid ways in which a function can be defined. Therefore this section will elaborate on function creation in JavaScript.

[1] https://facebook.github.io/react-native/

[2] https://insights.stackoverflow.com/survey/2016

2.2.1 Regular functions

A regular function is defined by making use of the `function` keyword like so:

```
1 function <name>(<parameters>) {
2     <statements>
3 }
```

The given `parameters` can be zero or more, and there are a variety of `statements` that can be used inside the body of the function. The function needs to be given a `name` and is called a **function declaration**.

Functions can also be assigned to variables or objects, which is considered a **function expression**. A function expression is structured the same way as a function definition, however, the name can be omitted. When no name is given, it is known as an anonymous function. Note that in both cases the function is called by using the name of the variable it is assigned to.

There are also generator functions available, which have the same syntax as a regular function with the exception of an asterisk (*) after the `function` keyword.

2.2.2 Function Constructors

Although not recommended, it is also possible to use a constructor to create a function. For this, the following syntax is used:

```
1 new Function(<params>, <body>)
```

Here names of the parameters are provided, and a string containing the body of the function. The created function can be saved to a variable for later usage. Again, there is a generator function syntax available, which is similar apart from the usage of `FunctionGenerator` instead of `Function`.

2.2.3 Other Function Constructions

If a function needs to be invoked immediately, instead of being able to call it, we can make use of an Immediately Invokable Function Expression (IIFE). An IIFE has the following format:

```
1 (function() {
2     <statements>
3 })();
```

This is often used for functions that are only called once, for example to initialize specific functionality when a website loads.

More recently, a different way to define a function was introduced, which is called an arrow function expression. An example of this can be found below:

```
1 (<params>) => { <statements> }
```

There are small changes to this syntax allowed. For example if a single parameter is used, the parentheses are not required. The same applies to a single statement; in that case it is not required to surround it by brackets.

2.2.4 Function Properties

As already stated in Section 2.2.1, functions can be assigned to variables or objects. When a name is given to such a function, it is called a **named** function, while omitting the name results in an **anonymous** function. One of the listed benefits of a named function is that it is shown in a stack trace in case any error occurs.

Due to the dynamic nature of JavaScript, functions do not define any return types. Additionally, there are no `public` or `private` modifiers available to limit access to certain functions. However, it can be achieved by making use of specific function constructs and closures in JavaScript, although this is rarely used in the projects that we looked into.

2.2.5 Parameters & Arguments

There is a difference between parameters and arguments in programming languages, as parameters refer to the names listed in a function definition, while arguments are the actual values passed to the function when it is called.

In JavaScript, it is allowed to call a function with less than the specified amount of parameters. In such a case, the missing parameters will default to 'undefined'. However, in recent versions of JavaScript a default value can also be assigned. Additionally, you can also call a function with more arguments than specified, which will then be available inside the function by making use of the `arguments` object. This approach is often used for functions where a variable number of arguments is expected, for example a function that concatenates multiple arrays. Instead of the `arguments` object, a rest parameter can also be specified for the function. In that case, the last parameter is specified as three dots (...) followed by a name. This rest parameter can then be accessed by using the specified name inside the function, giving access to the arguments in the form of an array.

As with functions, there are no types defined for the parameters.

2.3 Modularization with Node.js

Often seen in JavaScript projects, is the free server framework Node.js[3]. It enables JavaScript projects to consists of modules, and opens up some new possibilities that we should know of. The first being the ability to export certain functions that are defined in a JavaScript file. This enables outside access to only the selected functions, making it possible to recreate some sense of private functions that can only be used internally in the file. Furthermore there is a `require` statement that indicates that, part of, another JavaScript project will be included in the current file. Both these statements give way to new uses of functions, and should be kept in mind when looking for breaking changes. Examples of these constructs can be found in Listing 2.1.

[3] https://nodejs.org/en/

```
1 var myVar = require('./package'); //Imports the entire package as myVar
2 var myFunc = require('./package/func'); //Some projects make it possible
      to import a specific functionality
3
4 module.exports = myVar; //Exports myVar, making it available to outside
      files
5 exports.myProp = myVar; //Another way to export myVar
```

Listing 2.1: Node.js constructs

Chapter 3

An Approach to Detect Breaking Changes in JS Projects

This chapter presents our approach to detect breaking changes in JavaScript projects. A breaking change is defined as a part of a software system that causes other parts to fail. However, we make use of our own, stricter, definition in our research. Therefore, we consider a breaking change to be one of the following:

- A function that is removed between two versions
- A function that is renamed between two versions
- The number of parameters for a given function is changed between two versions

Furthermore, we omit breaking changes that have to do with types. Examples of this include changes in return types of functions or changes to parameter types. The reason being that we do not have type information in JavaScript. Even consulting frameworks such as Flow[1], to infer types, or TypeScript[2], to make use of premade ”type definitions”[3], did not provide us with enough information. In case of Flow, the majority of types could not be inferred in our projects, while the TypeScript files only contained some of the latest versions, omitting the majority of versions that we also make use of.

Our aim is to be able to list the breaking changes in consumer packages that depend on other JavaScript projects. Additionally we also look into the possibility of generating a list containing the breaking changes for each version of a project. A visualization of our approach to achieve this can be found in Figure 3.1, indicating the steps that are needed to get to our desired result. We start by transforming a project into an AST, which makes it possible for us to detect and extract functions. We are only interested in functions defined by the project, as those might change in a future version and introduce breaking changes. We generate a list of the relevant functions that we come across, and save this as our initial

[1] https://flow.org

[2] http://www.typescriptlang.org

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

breaking changes list. This list will consist of breaking changes in both public and private functions at this point.

Then we move on to the consumer packages, which we will use for an empirical study by looking for breaking changes in them, and it will also serve as a means to filter out private functions from our initial breaking changes list. Before we can achieve this, we first need to find all function invocations in each of the consumer packages. This, again, includes transforming the source files into ASTs. However, this time we go over all functions invocations and detect whether they make use of a project's function. If this is the case, we compare the found function against our list of breaking changes. If there is a match, we save the function as a breaking change for that consumer package, including additional information such as the file in which it was found and on which line. As a result, we end up with a list of breaking changes for each of the consumer packages. Since the consumer packages all made use of functions from the same project, we can combine the found breaking changes to produce a final list containing only breaking changes in public functions.

Each of the steps is explained in further detail in the resulting sections of this chapter.

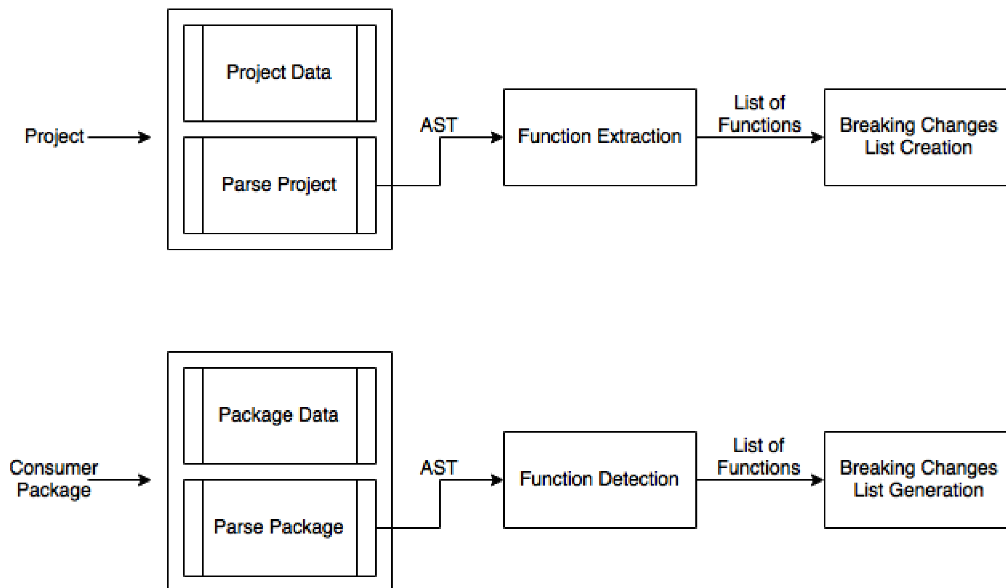


Figure 3.1: Approach Overview

3.1 Project Parsing

Summary: Transform the source code into a usable format for our tool

In order to parse the source code of a package, we first need to transform it into an AST. We looked into existing tools that can do this, as writing it from scratch would be too time

consuming and error prone. While looking into the available parsers, we also considered these requirements:

- Details such as line numbers should be included in the AST, as this information is relevant to the user to determine the exact location of a breaking change.
- It has support for React syntax[4], such as JSX elements and import statements.
- It makes use of Mozilla's standard parser API notation[5]. We can find extensive information on the constructs used in this standard API, which ensures that we can parse the right information from the generated AST. By adhering to a standard, it also opens op possibilities for future improvements as we do not rely on our own notation.

We can easily compare existing parsers with the help of an online tool called AST Explorer[4]. The website makes it possible to input code in a field and select one of multiple parsers that will then try to generate an AST from it. We randomly selected existing source code files from projects to see how the different parsers respond to the input. The results of this can be found in Table 3.1. Although the website does not contain an exhaustive list of all available parser, it does contain the most widely used ones and those that are still being maintained.

Name	**Notation**	**React**	**Line number**
Acorn	Standard	Plugin	No, only character count
Babel[6]	Standard	Yes	Yes
Esformatter	Standard, with extras	Plugin	Yes
Espree	Standard	No	No, only character count
Esprima	Standard	No	No, only character count
Flow	Standard	Yes	Yes
Recast	Standard	No	Yes
Shift	Slightly Modified	No	No
Traceur	Custom	No	Yes
Typescript	Slightly Modified	Yes	No, only character count
Uglify JS	Custom	No	Yes

Table 3.1: JavaScript Parsers

Out of the possible parsers that fit our requirements, we decided to go with Flow. The reason for choosing Flow over the other option, Babel[7], is due to the fact that Babel usually requires transforming (transpiling) source code in order to work with React elements. This extra step slightly rewrites the source code to comply with default JavaScript rules, which can introduce shifts in line numbers. Since we want to show the actual line numbers

[4]https://facebook.github.io/react/docs/jsx-in-depth.html
[5]https://developer.mozilla.org/en-US/docs/Mozilla/Projects/SpiderMonkey/Parser_API
[6]Babel includes three parsers: Babel-eslint, Babylon and Babylon6
[7]http://babeljs.io

when reporting breaking changes to the user, the transpiled code was not a good solution as additional origin tracking [23] would be necessary.

Furthermore, we need to look into the project versions that we consider. Since we are only interested in versions that are production ready, we created the following constraints:

The version is 1.0.0 or above Semantic versioning[8] considers versions that are of the form 0.x.x to be beta releases, which means that they are allowed to have breaking changes in every version change. Since we are interested in cases where semantic versioning is not adhered to in terms of breaking changes, we disregards versions below 1.x.x.

The version is a full version release With this we mean that the version is actually released and is not an alpha, beta or release candidate. In order to check for this, we do not allow versions that have a suffix (e.g. 2.0.0-rc) as that is generally the way to tag non-production versions.

3.2 Function Extraction

Summary: Detect functions in the generated ASTs and extract relevant functions that are defined by the project.

With Flow as our parser, we are able to transform a source code file into a corresponding AST. We look for any *.js files belonging to the selected project, disregarding any `node-modules` or other 3rd party folders. For each of the files that we find, we generate an AST with Flow and look for specific constructs to find any functions defined by the project. This comes down to looking for `FunctionExpression` and `FunctionDeclaration` keywords. We are only interested in functions that are directly defined by the project, meaning that we do not fully traverse the generated AST in depth. Instead, we only look into finding functions that are either directly defined in the file, are object properties or encapsulated inside an IIFE. An example can be found in Listing 3.1. Here we are only interested in the **throttle** function and need to discard the **later** and return functions, even though those are also valid instances of a `FunctionExpression`.

```
1  _.throttle = function(func, wait, options) {
2      var later = function() {...};
3      return function() {
4        //Makes use of 'later' function in here
5        return result;
6      };
7  };
```

Listing 3.1: A simplied example of a real-life Underscore function

[8] http://semver.org

Additionally, since we consider projects that work within a Node environment, we also look for `export` statements and the variable that is being exported by the file. This variable, or multiple variables, will also be seen as available functions, but only if they differ from the ones that we already found by looking at the `FunctionExpression` and `FunctionDeclaration` keywords and object properties.

Also, if there was an export statement found in a file, we disregard any functions that we found but where not exported. The reason for this is that only exported functions are available to outside files, which means that we can assume that any non-exported functions are private. Unfortunately this does not ensure that we end up with just public functions, as some internal functions are also exported as they are used throughout other files in the project. As a result, this means there can be functions defined in a project that are not documented as they are meant for internal use only. However, if a user really wants, it is possible to call these functions and make use of them in one's own package. In order to try to counter this issue, we will later in this Chapter explain how we plan to make use of consumer packages to filter out private functions.

For now, every time we come across a function that we expect to be defined by the project and available outside the file, we save its name and number of parameters to a list. This list is then saved to a file and given the name of the version of the project that we just checked. This step is then repeated for every eligible version of the project, resulting in a file with a list of encountered functions for each version. Since NPM does not provide an option to install multiple versions of the same project alongside each other, we make use of node-multidep[9] for this purpose.

3.3 Breaking Changes List Creation

Summary: Transform the extracted functions from a project into a list of breaking changes per version. Note that this results in a list that can still contain private functions and some public functions might be missed by the tool.

When the functions have been extracted for each eligible version of the project, we look for any differences between them to come up with a list of breaking changes. For this, we start with the first version (which we call `old`) of the project and compare it against the next version (which we call `new`). Every time a function was available in the `old` file, but can no longer be found in the `new` file, we consider it as a breaking change between those two versions. The same applies to changes in parameters, if a function in the `old` file has a different amount of parameters than the `new` file, it is also considered as a breaking change.

For each of the breaking changes that we find, we save the function name and the type of breaking change. A breaking change type can fall in one of the two following categories:

[9] https://github.com/joliss/node-multidep

Removed When a function is either renamed or removed from the code, we mark it as removed. The reason for letting renamed functions fall into this category is based on the fact that we cannot detect renames in the source code.

Changed When the number of parameters for a function differs between versions, we mark it as changed.

In case of the *changed* type, we also include the amount of parameters that it had in the `old` and `new` version.

The reason for choosing to work with categories has to do with the loosely typed nature of JavaScript. Since we have no reliable way to detect whether a parameter for a function is optional, a change in parameters might not necessarily result in a breaking change. There are cases in which an optional parameter is used internally, while the API does not mention it being used. When such a parameter is removed, the code will be changed while it is not an actual breaking change. By using categories, users can distinguish between severity of the breaking changes, as removals will definitely affect them while changes in parameters might not always be an issue.

Since we are interested in the breaking changes between different versions, this process is repeated several times. Each time and `old` version is compared to a later `new` version, so we end up with a file of breaking changes for each of the later versions. To give a more concrete example, we will have a list of breaking changes between version 1.0.0 and every later version, v2.0.0 and every later version, and so on. Note that these lists can still contain private functions and that some public functions, such as the example discussion in Section 6.1, might be missed by the tool.

3.4 Consumer Package Parsing

Summary: Consumer packages will be used for two reasons: finding breaking changes in a consumer package, and to filter out private functions from our list of breaking changes for a project. In order to do this, we first need to change the source code of the consumer packages into a usable format and extract the version of the project the consumer package makes use of.

Now that the breaking changes have been found the project, we will look into the consumer packages to filter out the private functions in the list. For this, we install a consumer package through NPM and determine the version of the project that they make use of. This can be done by inspecting the `package.json` file in the package. There are different ways in which the project versions are used, therefore we made the following criteria to select the right version:

x.x.x When a specific version is defined by the package, we will simply make use of that version. *Example:* 2.1.0 or 1.14.8.

x.x.x with prefix According to semantic versioning, it is possible to define ranges for versions. If a package has defined a range, which can be seen by the version number having a prefix of a specific character, we choose the first version in this range. *Example:* ˆ4.0.0 or ˜2.1.0

*** or ''** Indicates that any version can be used. If this is the case, we make use of the earliest version of the package, which is typically 1.0.0, to find breaking changes for.

latest Indicates that the latest version of the package should be used. In that case we also make use of the earliest version of the package as we do not know what version would be considered *latest* at the time the package was created.

There are also cases in which the version did not meet one of the aforementioned criteria. In those cases, it mostly consisted of an incorrect version format due to human error. In those cases, manual inspection was needed to correct the version to make it adhere to one of the listed criteria.

After the used version has been established, Flow is again used to generate an AST of the *.js files in the package. Again, we disregarded any `node-modules` and 3rd party folders used by the package.

3.5 Function Detection

Summary: Find all function invocations in the AST of the consumer package and extract relevant ones belonging to the project for which we check for breaking changes.

As long as the project version that is used by the consumer is not the latest version, we generate and traverse the ASTs and search for all function invocations that are available. Additionally, we check for any `require` statements to ensure that the file makes use of any functions from the project. If no `require` statement is found in the file, we move on to the next as we can assume that no functions from the project will be available in the AST anyway. When a `require` statement is found, we check whether it is referring to a single function or the project in a whole. In both cases, we save the variable name in which the function or project is saved and go over the AST to look for any function invocations. Whenever a function invocation is found, we check this against the variable to see whether we found a function that was defined by the project. If this is the case, we check whether the function was listed in any of the breaking changes files for specific version the consumer makes use of. If we find a match, we save the function as a breaking change for that particular consumer package. In addition to the function name, we also save the file name, line number and type of breaking change. This data will make it easier for the consumer to pinpoint the location in its source code where the breaking changes occurred.

3.6 Breaking Changes List Generation

Summary: For each of the consumer packages, we compare the list of functions that we found against the initial list of breaking changes that we created. Each function that is available in both of the lists, is considered a breaking change in a public function and saved accordingly.

When we went over all the files in a consumer package, we end up with a list of breaking changes that the consumer will run into when trying to upgrade to a later version of the used project. Since a single consumer will most likely not use all of the functions that a project provides, we repeat the previous step for multiple consumer packages. This results in a list of found breaking changes for each consumer package.

After the 1000 consumer packages have been checked and their respective breaking changes files have been created, we combine them. That way, we will end up with breaking changes for each of the versions of a project, without any private functions as we assume that consumers only use the documented public ones.

Additionally, we also have a list of breaking changes for each of the individual consumer packages.

Chapter 4

Methodology

The goal of this chapter is to explain how we evaluate the performance of the tool. Furthermore, we are interested in the effect breaking changes have on JavaScript packages. Therefore we came up with the following research questions to guide our evaluation:

RQ1: How accurate is the tool in detecting breaking changes? We are interesting in finding out how reliable the tool is and if there are cases in which it performs better than in others. From this we can determine whether the tool is generally applicable to JavaScript projects.

RQ2: In what cases does the tool produce incorrect results? If the tool incorrectly diagnoses changes as breaking changes, we want to know what the underlying reasons are for these errors. This will show whether there are limitations in the approach or shortcomings in the current implementation of the tool. Furthermore, it will give an insight in what cases the tool will perform to a lesser extend.

RQ3: How often are projects affected by breaking changes? Instead of solely focusing on the tool and its performance, we also want to know how often breaking changes occur in projects. In case of semantic versioning, we would expect a project to only contain breaking changes between major version changes and we're interested to see whether this holds for our selected projects.

RQ4: To what extent do breaking changes affect consumer packages? Whether the consumer packages are affected by breaking changes is also something to consider. We will look into the frequency of encountered breaking changes, which will indicate whether the tool could be a valuable asset for developers. Additionally, we are interested in seeing whether packages are often kept up to date or whether there are still a lot of outdated projects that contain breaking changes.

4.1 Project Selection

To properly evaluate the tool we created, a selection of JavaScript projects and consumers of those projects are necessary. Therefore we turned to the NPM Registry[1] website and looked for the "most depended upon" projects [2], as that indicates their popularity and ensures recent usage of it. Additionally, due to the many users that trust on them, we expect them to be a good example of how a JavaScript project should be in terms of maintaining code and stability. From the web page we selected the following three projects:

Lodash At the time of writing, the most depended upon project is Lodash with over 50 thousand dependents. Lodash is meant to enhance JavaScript with often used functionality that is not available out-of-the-box. It contains all kinds of utility functions, and aims to be versatile and lightweight, making it fit in almost any project. Due to its modular nature, it is not necessary to include the full Lodash package. Instead, it is possible to include only the precise functionality that is needed.

Underscore The second project we chose is Underscore. Underscore is actually the older brother of Lodash, as Lodash started as a fork of it. However, the two projects have a vast different approach to writing code and focus on different aspects. We chose this project because it is still decently popular with over 16 thousand dependents and its coding style differs from Lodash.

Bluebird Bluebird is the last project we chose. It differs from the other two project, as it doesn't aim to be a utility project with a lot of different functionality, but instead provides a new `Promise` class with its own functionality. This project poses new challenges as types are not readily available in JavaScript and inferring them is difficult.

For each of the aforementioned projects, we created a list of versions that were to be considered by the tool. This list only includes release-ready versions (version $\geq$ 1.0.0), meaning that no alpha, beta or release candidate versions are considered. This version information is retrieved from the NPM registry. A full overview of the considered versions per project can be found in table 4.1.

Furthermore, we need a set of consumer packages that depend on the project to evaluate the usability of our tool. Therefore we looked at the "dependents" page of a project[3] and selected the first 1000 distinct consumer packages. Although the exact ordering of this page remains unclear[4], we believe it to be ordered by the package that most recently requested the project.

[1] https://www.npmjs.com
[2] https://www.npmjs.com/browse/depended
[3] https://www.npmjs.com/browse/depended/lodash
[4] https://github.com/npm/www/issues/152

Project	No. of Versions	Versions
Lodash	73	1.0.0, 1.0.1, 1.0.2, 1.1.0, 1.1.1, 1.2.0, 1.2.1, 1.3.0, 1.3.1, 2.0.0, 2.1.0, 2.2.0, 2.2.1, 2.3.0, 2.4.0, 2.4.1, 2.4.2, 3.0.0, 3.0.1, 3.1.0, 3.2.0, 3.3.0, 3.3.1, 3.4.0, 3.5.0, 3.6.0, 3.7.0, 3.8.0, 3.9.0, 3.9.1, 3.9.2, 3.9.3, 3.10.0, 3.10.1, 4.0.0, 4.0.1, 4.1.0, 4.2.0, 4.2.1, 4.3.0, 4.4.0, 4.5.0, 4.5.1, 4.6.0, 4.6.1, 4.7.0, 4.8.0, 4.8.1, 4.8.2, 4.9.0, 4.10.0, 4.11.0, 4.11.1, 4.11.2, 4.12.0, 4.13.0, 4.13.1, 4.14.0, 4.14.1, 4.14.2, 4.15.0, 4.16.0, 4.16.1, 4.16.2, 4.16.3, 4.16.4, 4.16.5, 4.16.6, 4.17.0, 4.17.1, 4.17.2, 4.17.3, 4.17.4
Underscore	33	1.0.3, 1.0.4, 1.1.0, 1.1.1, 1.1.2, 1.1.3, 1.1.4, 1.1.5, 1.1.6, 1.1.7, 1.2.0, 1.2.1, 1.2.2, 1.2.3, 1.2.4, 1.3.0, 1.3.1, 1.3.2, 1.3.3, 1.4.0, 1.4.1, 1.4.2, 1.4.3, 1.4.4, 1.5.0, 1.5.1, 1.5.2, 1.6.0, 1.7.0, 1.8.0, 1.8.1, 1.8.2, 1.8.3
Bluebird	125	1.0.0, 1.0.1, 1.0.2, 1.0.3, 1.0.4, 1.0.5, 1.0.7, 1.0.8, 1.1.0, 1.1.1, 1.2.0, 1.2.1, 1.2.2, 1.2.3, 1.2.4, 2.0.2, 2.0.3, 2.0.4, 2.0.5, 2.0.6, 2.0.7, 2.1.1, 2.1.2, 2.1.3, 2.2.0, 2.2.1, 2.2.2, 2.3.0, 2.3.1, 2.3.2, 2.3.3, 2.3.4, 2.3.5, 2.3.6, 2.3.9, 2.3.10, 2.3.11, 2.4.0, 2.4.1, 2.4.2, 2.4.3, 2.5.0, 2.5.1, 2.5.2, 2.5.3, 2.6.0, 2.6.1, 2.6.2, 2.6.3, 2.6.4, 2.7.0, 2.7.1, 2.8.0, 2.8.1, 2.8.2, 2.9.0, 2.9.1, 2.9.2, 2.9.3, 2.9.4, 2.9.5, 2.9.6, 2.9.7, 2.9.8, 2.9.9, 2.9.10, 2.9.11, 2.9.12, 2.9.13, 2.9.14, 2.9.15, 2.9.16, 2.9.17, 2.9.18, 2.9.19, 2.9.20, 2.9.21, 2.9.22, 2.9.23, 2.9.24, 2.9.25, 2.9.26, 2.9.27, 2.9.28, 2.9.29, 2.9.30, 2.9.31, 2.9.32, 2.9.33, 2.9.34, 2.10.0, 2.10.1, 2.10.2, 2.11.0, 3.0.0, 3.0.1, 3.0.2, 3.0.3, 3.0.4, 3.0.5, 3.0.6, 3.1.0, 3.1.1, 3.1.2, 3.1.3, 3.1.4, 3.1.5, 3.2.0, 3.2.1, 3.2.2, 3.3.0, 3.3.1, 3.3.2, 3.3.3, 3.3.4, 3.3.5, 3.4.0, 3.4.1, 3.4.2, 3.4.3, 3.4.4, 3.4.5, 3.4.6, 3.4.7, 3.5.0

Table 4.1: Considered versions per project

4.2 Creating the Oracle

For each of the selected projects in Section 4.1, we created our oracle of breaking changes. This was done manually by looking through the provided documentation, additional breaking changes information and manual code inspection. The oracle contains information on the version in which the breaking change was found, and its type (e.g. removed or changed). This information is saved in the same fashion as the results our tool produces and will give us insights in the performance of the tool. Because of the code inspection that was sometimes needed due to lacking documentation, the creation of the three oracles took around a week in total.

An oracle contains all of the breaking changes for a given project. However, since we

use our 1000 consumer packages to evaluate the performance of the tool, the oracle can contain functions that were not used in our selection of consumer packages. This will have an influence on the results, as our tool will not be able to report on any unused functions. Therefore we try to remove any of the unused functions from our oracle by deploying a text search strategy. This means that for all functions in our oracle, we do a text search in the 1000 consumer packages to see if it is used. If the result comes back negative, we remove the specific function from the list to increase the reliability of our analysis. For the text search, we consider the following patterns:

functionName(The most obvious function usage is a regular invocation of the form `function(parameters)`. Since the parameters can vary, we search for just the function's name, followed by an opening parenthesis. Also note that there is a space in front to ensure that we match the full name of the function and not as part of another function. *Example:* find() but not quickFind().

.functionName(This pattern is similar to the previous one, but instead of a space at the front we have a dot. This is to match cases where functions are and object property. *Example:* myObject.find().

project/functionName Since we consider packages that work with Node, a function can also be included from a project and assigned to a variable. To cover this possibility, we search for the existence of `project/functionName`. *Example:* require lodash/find

When a given function does not match any of the above patterns, we assume that it is not used by any of the consumer packages and remove it from the oracle. However, a text search is prone to error and could report a match even though it should not. To give some examples of this:

- A different function in the consumer's source code can have the same name as the one we are looking for. Even though the text search would produce a result, it is not the actual function we were looking for.
- A function can be part of a comment, which the text search does not distinguish from valid code.

The above cases show that it is possible that the oracle still contains functions that are not used in the consumer packages. In case this happens, the precision and recall rates we calculate can never be worse in practise. The reason for this is that if any function that was incorrectly listed would be removed from the oracle, it will only have a positive effect on the values we calculated. To ensure that the text search produces the results we expect, we manually looked into 10 random consumer packages to validate its findings.

Additionally, there could be cases in which a function is very rarely used (e.g only once in the 1000 consumer packages) and be missed by the tool. In that case it will have an influence on the precision and recall in theory, while in practice this function is almost never used and the actual precision and recall of the tool will be higher for most consumers. Again, this means that the results will not be worse than shown in our results in Chapter 5.

4.3 Data Analysis

We analyze the data with two perspectives in mind. The first one addresses the ability to find the breaking changes between project versions, while the other focuses on the consumer side and how useful the tool can be to them in determining breaking changes in their code.

4.3.1 Project Aspect

With the data retrieved from running the tool on the 1000 consumer packages, we calculate the precision and recall for each of the projects. This is done by comparing the results from the tool one-on-one with the oracle that we created. This means that a breaking change found by the tool is only considered correct if the function name and its breaking changes type match the ones listed in the oracle.

Whenever a function is incorrectly listed by our tool as a breaking change, we manually inspect the underlying reason of this error. These reasons can be divided into five categories, which can be found in more detail in Section 5.2. We keep track of the amount of functions that fit in each of the categories to get an idea in which cases the tool performs less.

4.3.2 Consumer Aspect

For the consumer part, we are not interested in all the possible breaking changes in our oracle. Instead, we want to know which breaking changes occur in a specific code base, as those are relevant to a consumer, and how well the tool performs in that case. Therefore we will use the individual breaking changes lists that we generated for each of the consumer packages.

Since the tool reports breaking changes related to the current version of the consumer package, it happens that the same breaking change is listed for later versions (e.g 2.0.0, 2.0.1, 2.0.2, etc.) Since the oracle will only list this breaking change for version 2.0.0, we first need to ”normalize” the consumer's breaking changes list. For this we discard the breaking changes that were already available in an earlier listed version, leaving only the version in which the breaking change was introduced.

Next, we compare the normalized list against our oracle to see how the tool performs. Here we keep track of the following metrics:

Correct These are the functions that appear in both the consumer list and our oracle.

Incorrect These are the functions that only appear in the consumer list and are therefore incorrect breaking changes.

Missed These are the functions that were not found by our tool, but were listed by the oracle.

Generally, a consumer will not make use of all the functions a project defines. When a function is not used, it will be considered as ”missed” by our tool, as it will not be able to find it. This is not a good representation of the actual performance of the tool. Therefore we want to filter out these functions from the oracle, to get a more reliable result. We achieve this by

using the same text search as explained earlier in Section 4.2 and discard any functions that are not found, as can be seen in figure 4.1. That way we create an individual oracle for each of the consumer packages, tailored to the functions that they make use of. In our example of Figure 4.2, this means that version 2.0.0 of a project has a total of 9 breaking changes according to the oracle we constructed. We check for each of these 9 functions whether they were used in the consumer package, and discard them if they were not. As a result, only **function1** is used in the consumer package, while the remaining 8 functions that contain breaking changes were not. This results in our consumer package oracle containing only **function1**, which we can then compare with the breaking changes that we found. Do note that also in this case the text search is not perfect and can influence our results. However, they will never be worse than what we report.

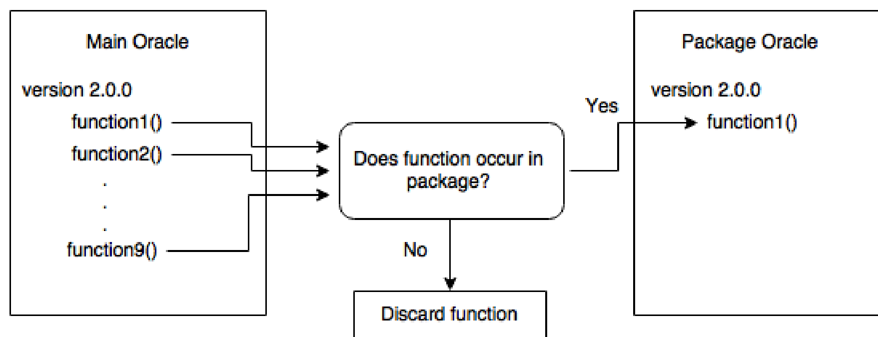


Figure 4.1: Creating a consumer package oracle

Finally, the resulting data is used to determine the precision and recall for each of the individual consumer packages. For the recall calculation we make use of the amount of missed functions, minus the amount of functions that were discarded, to get a closer-to-reality result.

Chapter 5

Results

This chapter covers the results of the project. Sections are divided into separate research questions, each presenting the relevant results to that question.

5.1 RQ1: How accurate is our approach in detecting breaking changes?

In total, the tool was applied on three different JavaScript projects, for which we looked at 1000 consumer packages each. We cover the results per project in seperate sections.

5.1.1 Lodash

As can be seen in Table 5.1, the tool returns a total of 114 breaking changes in the case of Lodash. Of these, 64 correspond to actual breaking changes that we found by manual inspection, giving it a precision of 0.56. This indicates that there were 50 incorrect functions found by the tool. The 64 breaking changes found by the tool is less than the actual 96 breaking changes that are available, giving the tool a recall of 0.68.

# changes	# changes found	Correct	Precision	Recall
96	114	64	0.56	0.68

Table 5.1: Lodash breaking changes

5.1.2 Underscore

Due to the lack of a full breaking changes list, we decided to pick a subset of versions for which we check the found changes by the tool against manually inspecting the version. The results of this can be seen in Table 5.2. The table also shows the aggravate of the five chosen versions in order to get a final precision and recall rate. Due to the high amount of false positives, the precision rate remains low at just 0.22. However, the tool did not miss many of the breaking changes, making its recall as high as 0.80 for this project.

Version	Manually	Tool	Correct	Missing	Precision	Recall
1.8.3	0	1	0	0	0.0	1.0
1.8.0	2	7	2	0	0.29	1.0
1.7.0	2	6	2	0	0.33	1.0
1.6.0	0	1	0	0	0.0	1.0
1.4.0	1	3	0	1	0.0	0.0
Total	*5*	*18*	*4*	*1*	*0.22*	*0.80*

Table 5.2: Underscore breaking changes

5.1.3 Bluebird

Table 5.3 shows the precision and recall rates for a selection of versions of Bluebird. There were not that many breaking changes available in the project, apart from the major versions, which is a first. Because of this, the precision and recall rates are both 1.0 in two versions, as there were no breaking changes and the tool also reported none. Overall, the precision comes down to just 0.3, mostly due to private parameters being removed between versions. Furthermore, the recall rate is 0.43, which we can address to the missed breaking changes.

Version	Manually	Tool	Correct	Missing	Precision	Recall
3.4.0	0	0	0	0	1.0	1.0
3.0.0	4	6	3	1	0.5	0.75
2.4.2	0	0	0	0	1.0	1.0
2.0.2	3	1	0	3	0.0	0.0
1.2.0	0	3	0	0	0.0	1.0
Total	*7*	*10*	*3*	*4*	*0.3*	*0.43*

Table 5.3: Bluebird breaking changes

5.1.4 Consumer Packages

Furthermore, the precision and recall is calculated for the individual consumer packages. The results of this can be seen in Table 5.4. From the data it follows that the precision and recall are generally better for the individual consumers, with an exception of recall for Lodash (0.28 vs 0.66) and Underscore (0.77 vs 0.80). Note that these results are only based on consumer packages that contained at least one breaking change. Also, the precision and recall in this particular case are calculated by using the filtered oracles as explained in Section 4.3.2.

RQ1: For the total amount of breaking changes per project, the precision ranges from 0.22 to 0.56 and the recall sits a bit higher between 0.43 and 0.80. For consumer

Project	Precision (Mean)	Recall (Mean)
Lodash	0.62	0.28
Underscore	0.59	0.77
Bluebird	0.82	0.77

Table 5.4: Precision and Recall for the individual packages

packages, the mean precision ranges from 0.59 to 0.82 and the recall is between 0.28 and 0.77.

5.2 RQ2: In what cases does the tool produce incorrect results?

Since the tool is not perfect in terms of finding breaking changes, we looked into the reasons for any errors that it made. We were able to divide these errors into the following categories:

Arguments.length Instead of defining the parameters a function takes, `arguments.length` is often used. A possible reason for this is that it makes it easier to work with variable-length functions, such as a `zip` function that can take multiple arrays. However, often the initial version of such a function did take one or more parameters, after which a change was made to `arguments.length` in a later version. Since the parameters has been removed from the function, the tool informs us about the amount of parameters being changed between versions. Unfortunately this does not necessarily mean it's a breaking change for the user, as the function could still take the same amount of parameters as before at runtime.

Function return There are cases in which a previously defined function is changed to call another function. That function then returns an object containing the actual function, based on some given parameters. However, the tool can only do a static analysis of the code, which means that it does not know what kind of function is returned in this case. Because of that, changing from a regular function definition to a function that returns the actual function, is seen as a breaking change as the code involved often changes in terms of parameters.

Private parameters In the case of JavaScript, it is not possible to define parameters as public or private. Because of that, functions sometimes contain a parameter, often the last one, that is only used internally and is not listed in the official documentation. These "private" parameters can be removed, without negative effects on the users as they should not have been used by them. However, the tool cannot distinguish between public and private functions, which results in these type of changes being listed as breaking changes.

Tool incorrect These are the cases in which the tool incorrectly listed a function as a breaking change. In other words; the tool made a mistake. The exact reasons vary and are

not always clear, therefore making it difficult to pinpoint the issue in the code. As a results, these issue currently remain for future updates of the tool as we were unable to resolve them at the time of writing.

For each of the projects, the incorrect functions were manually inspected and classified accordingly. Table 5.5 shows the result of this classification for the three projects combined.

Reason	**Amount**	**Percentage**
Function Return	27	38.6%
Private Parameters	22	31.4%
Arguments.length	13	18.6%
Tool Incorrect	8	11.4%

Table 5.5: Classification of incorrect breaking changes

From the data it follows that the majority (almost 90%) of incorrect breaking changes are due to changes in the code. This is topped by **function returns** in 38.6% of the cases, closely followed by the removal of **private parameters** with 31.4%. The **arguments.length** are less often encountered and therefore make up 18.6% of the cases, which is still a large amount. Only 11.4% is due to mistakes by the tool.

RQ2: Around 70% of the incorrect breaking changes are due to private parameters or changes to function returns. Another 18% is due to making use of arguments.length instead of regular parameters.

5.3 RQ3: How often are projects affected by breaking changes?

Other than the amount of breaking changes that we find, it is also interesting to look at how often breaking changes occur and in which versions. Tables 5.6, 5.7 and 5.8 show the amount of breaking changes that were found for the different packages. For each version type in terms of semantic versions, which means `patch`, `minor` or `major`, the table shows the amount of breaking changes that were found by manual inspection and when running the tool. Note that in case of Underscore there are no major versions available, as the first available version on NPM starts at 1.0.3 and the current version is 1.8.3. For the Bluebird packages we considered version 2.0.2 to be a major version release, as it contained hotfixes and the actual version 2.0.0 was not made available on NPM.

Version	**Manual**	**Tool**
Patch	0 (0%)	3 (3%)
Minor	9 (9%)	36 (43%)
Major	92 (91%)	44 (53%)

Table 5.6: Lodash amount of breaking changes per version type

Version	Manual	Tool
Patch	6 (40%)	6 (25%)
Minor	9 (60%)	18 (75%)
Major	0 (0%)	0 (0%)

Table 5.7: Underscore amount of breaking changes per version type

Version	Manual	Tool
Patch	0 (0%)	0 (0%)
Minor	0 (0%)	4 (36%)
Major	7 (100%)	7 (64%)

Table 5.8: Bluebird amount of breaking changes per version type

From the tables it follows that projects have a different approach to handling breaking changes in versions. If we take a look at the manually found breaking changes for Bluebird, we see that 100% of the breaking changes are found in major versions, which is 91% for Lodash and an interesting 0% for Underscore. For both Lodash and Bluebird the tables show 0% of breaking changes to be found in patch versions, although Lodash does have 9% of breaking changes in minor versions. In case of Underscore this number is a lot higher, as 40% of breaking changes are found in patch versions.

When considering the results obtained with the tool, the general consensus is the same. Although the exact numbers differ from the manual analysis and the differences between versions appear to be smaller, the order in which they are presented is the same. For example, the tool shows that the majority of changes for Underscore are in minor versions (75%), while a smaller amount is found in patch versions (16%), which corresponds to the manual findings.

Table 5.9 shows the total amount of versions and the corresponding amount of versions with breaking changes per project. From the table we see that the projects that have the majority of breaking changes in major versions, have a smaller amount of versions that contain breaking changes overall. This ranges from just 1.6% in case of Bluebird to 9.6% in case of Lodash. Underscore on the other hand contains breaking changes in 36.3% of its versions.

Project	Versions	Versions with breaking changes	No. of breaking changes
Lodash	73	7 (9.6%)	96
Underscore	33	12 (36.3%)	15
Bluebird	125	2 (1.6%)	7

Table 5.9: Amount of versions per project with breaking changes

RQ3: Most of the breaking changes occur in major version changes, except for Underscore where it occurs in minor version changes instead. The overall amount of versions that contain breaking changes range from 1.6% to 36.3%.

5.4 RQ4: To what extent do breaking changes affect consumer packages?

Other than the versions in which breaking changes occur, we were also interested in the distribution of versions used by current consumer packages. In Figure 5.1 we can see the distribution of versions for consumer packages. It starts with the amount of packages that are up to date, followed by indicators of how far behind a consumer package falls. To give an example of this; if the latest version of a project is 4.1.7, a patch is anything between 4.1.0 and up, while a minor version is anything between 4.0.0 and 4.1.0 and a major version is everything below 4.0.0, with an exception of beta releases (typically below 1.0.0). In case of 'Unknown' there was not a specific version specified for the consumer package.

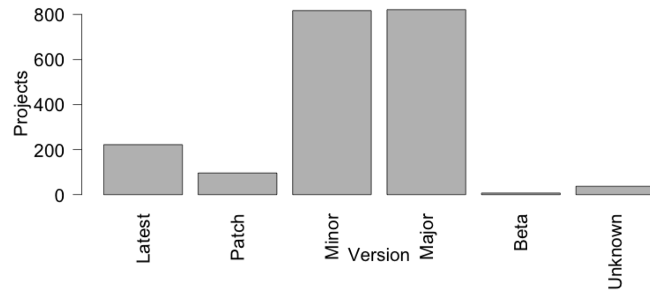
From Figure 5.1 it follows that the majority of consumer packages are not up to date with the latest version. In case of Lodash, Figure 5.1a, 11% of packages is fully updated and for Bluebird, Figure 5.1c, this is only 7.6%. Underscore on the other hand has the largest amount of up to date consumer packages, which is 32% as can be seen in Figure 5.1b.

Another important observation is that a large part of the consumer projects is at least a major version behind. In case of Lodash this accounts for 41% of the packages and Bluebird follows with 30%. Although Underscore shows 0% of packages being a major version behind, 54% of its consumers are at least a minor version behind. Due to the way in which Underscore list versions, with the absence of major ones, we can consider being a minor version behind to be of the same importance as a major version in the other two.

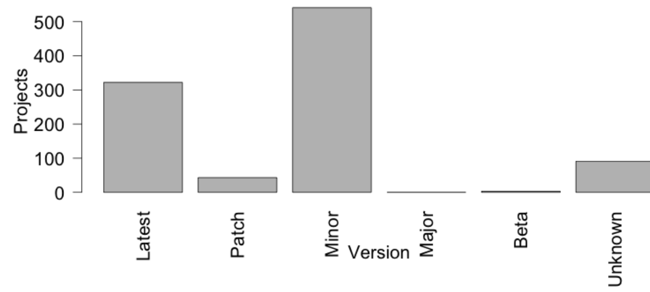
Additionally, we looked into the amount of consumer packages that contain breaking changes. Note that the results in Section 5.1.4 report that the precision for individual consumer packages ranges from 0.59 to 0.82 between the project. This means that between 18% and 41% of the breaking changes we found are false alarms, which can result in the actual amount of consumer packages containing breaking changes being lower than reported here.

From Table 5.10 it follows that of the 1000 consumer packages we looked into for Lodash, we found breaking changes in 25.8% of them. For Underscore this is 11.4% and in case of Bluebird it is even lower at 9.8%. The amount of packages that make use of the latest version ranges from 7.6% for Bluebird to 32.2% for Underscore.

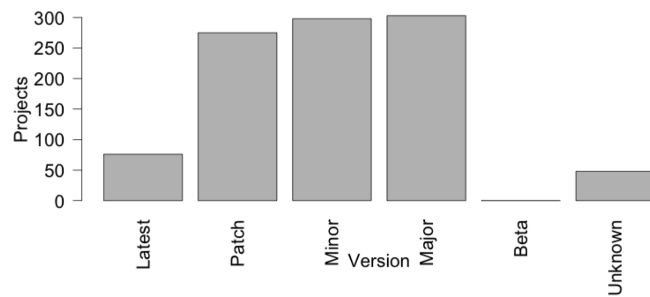
RQ4: Between 7.6% and 32% of consumer packages make use of the latest version of a project, indicating that the majority of projects are not up to date. Furthermore, the percentage of consumer packages that suffer from breaking changes ranges from 9.8% to 25.8% between projects.



(a) Lodash



(b) Underscore



(c) Bluebird

Figure 5.1: Version distribution for consumer packages

Project	Up to date	Without Breaking Changes	With Breaking Changes
Lodash	161 (16.1%)	581 (58.1%)	258 (25.8%)
Underscore	322 (32.2%)	564 (56.4%)	114 (11.4%)
Bluebird	76 (7.6%)	826 (82.6%)	98 (9.8%)

Table 5.10: Number of consumer packages per project that are up to date, without breaking changes and with breaking changes

Chapter 6

Discussion of our Findings

In this Chapter we evaluate our results of Chapter 5. Furthermore it discusses known limitations of our approach and possible improvements that can be made. Threats to validity are considered and the chapter ends with suggestions for future work.

6.1 Approach Accuracy

From the results in Section 5.1 and 5.2 it follows that there is still room for improvement of our tool. There are a number of known limitations that we will discuss next, including possible solutions if we are aware of any.

Dynamic Function Allocation

To start with, there are creative function constructs used in projects due to the dynamic nature of JavaScript. Because of this, there are possibilities of declaring functions at runtime, which our static analysis tool will not discover. An example of this can be found in Underscore, of which the respective source code can be found in Listing 6.1.

```
1 // Add some isType methods: isArguments, isFunction, isString, isNumber,
    isDate, isRegExp, isError.
2  _.each(['Arguments', 'Function', 'String', 'Number', 'Date', 'RegExp',
     'Error'], function(name) {
3    _['is' + name] = function(obj) {
4      return toString.call(obj) === '[object ' + name + ']';
5    };
6  });
```

Listing 6.1: Creative function creation in Underscore

Here we see an array of function names on line 2 that are assigned to properties of a global Underscore object on line 3. These properties are given the name `is`, followed by one of the function names that were defined on line 2. The actual body of the function is then shown on line 4. In this example we see that 7 functions are dynamically being defined, which in turn means that our tool misses these 7 functions and impacts our results.

Types
Although not available in standard JavaScript, there are projects that extend the language to make it possible to define types. Two well known examples of this are TypeScript[1] and Flow[2], backed by major companies such as Microsoft and Facebook. Both come with annotations to define types for function, parameters and more. For our research we also looked into these two programs, as Flow has a type inference ability while TypeScript makes it possible to provide Type Files[3] that can be used for existing JavaScript projects. Both options were tried on our selection of projects, but did not produce any useful results. In case of Flow, the inference option did not uncover the majority of the available types, adding no new information for our tool to work with. In case of TypeScript, there are some type files available for the projects, but those are only made for the last couple of versions. Since we were interested in breaking changes from the initial version to the current one, this did also not provide enough information to work with. However, we are hopeful that these projects will improve further over time. This might mean that more and more projects will start to use types in JavaScript, opening up the possibility of extending the tool with more functionality, such as checking for changes in return types of a function.

Function Body Changes
Currently the tool focuses on function signatures, while changing just the body of a function can also result in a breaking change. If there would be a change to what a specific function calculates, part of a program that assumes the old workings of a function will behave unexpectedly or even throw an error. At the moment the tool is not aware of such changes. Therefore it is interesting to look into the possibility of taking the body of a function into account and look for differences in number of lines, or even keep track of the actual body and compare those between versions.

6.2 Performance of the Tool

The installation of the consumer packages through NPM takes up a lot of time. In our case, the tool was run on a machine in Eindhoven, which is part of the Big Software on the Run[4] project. This machine has multiple times more processing power than the average computer, which already improved the installation process. Further improvements are discussed below:

Progress bar First of all, the progress bar seems to slow down the installation of packages through NPM[5]. However, this should already be fixed in the later versions of Node

[1] http://www.typescriptlang.org
[2] https://flow.org
[3] https://github.com/DefinitelyTyped/DefinitelyTyped
[4] http://fmt.ewi.utwente.nl/research/projects/3TU.BSR/
[5] https://github.com/npm/npm/issues/11283

and when we disabled the progress bar it did not result in noticeable differences, as expected.

Registry Another possible solution to improve performance, is to change the default registry, which is https, to its unsecured http variant.[6] In order to do this, we can use the command `npm config set registry http://registry.npmjs.org/`, after which we cleared the cache and tried installing the packages again. Unfortunately, this also did not result in noticeable differences.

Cache The last change that seemed promising is by making use of the built-in npm cache. This cache is only 10 seconds by default, but we can configure it by using the `--cache-min` option, followed by the time to keep packages in the cache in seconds.[7] Since the installation of the packages will take a decent amount of time, we decided to go for `--cache-min=9999` and tried to install the packages again. The cache option did not provide useful results, as the time needed to install 200 packages was still over three hours. Later on we also learned that there can be unwanted side effects of the cache, since npm will fail to install a package if the cache version does not match the requested version of a package. Even though we did not run into this issue ourselves yet, it is better to avoid it at all.

Yarn Yarn [8] can also be used to install packages, and claims to be fast at it too. Since we were looking for something to increase performance, we also gave this a try. We installed Yarn on the machine and used it to install the first group of packages. It started of quickly, just like NPM, but eventually had the same issue as installing a package started to take longer as the node_modules folder started to grow. Eventually, after 75 minutes, Yarn had installed 140 packages but it unexpectedly stalled after that. Although Yarn was initially faster than NPM, it suffers from the same issue where it grows slower over time when more packages are already installed.

Folders Since we saw the performance degrade when more packages were installed, we were curious as to what would happen if we divide a group of 400 packages into even smaller chunks and install those in separate folders. Therefore we created a script that divides a group of packages into 10 smaller parts, each consisting of 40 packages. We then create a new folder for such a part, in which we install the 40 packages through NPM. This is repeated until all parts of a group have been installed in their separate folders. As expected, the first couple of packages were installed quickly, after which the installation slowed down. It took approximately ten to fifteen minutes to install a single part. However, when we moved on to the next part and started installing the packages in a new folder, the performance was up again. In total it took only 35 minutes to install 200 packages this way, which is a huge improvement over the earlier 5 hours for just 200 packages. By running this script in parallel for

[6] https://github.com/npm/npm/issues/11028
[7] https://docs.npmjs.com/misc/config
[8] https://yarnpkg.com/lang/en/

the remaining groups, we were able to install all of the remaining consumer packages in just over 40 minutes.

Now that we are able to quickly install the required consumer packages, only combining the different folders remained. As we want to have all the packages belonging to a group in a single node_modules folder, we created a script that moves the separate parts to a combined folder, which only takes a couple of seconds to complete.

6.3 Threats to Validity

Several issue arise that can threaten the validity of this research. We start with the *internal validity* of the study, which is concerned with possible errors or bias in our research. We account the oracle that we created be part of this, as it can be flawed. There might be breaking changes that were not properly documented or that failed to grasp our attention when going over parts of the source code.

Additionally, the selection of consumer projects is based on the NPM web page. As we do not know the exact ordering of the page, the selected projects might not be entirely random. Therefore the selection could be biased and might not be representative of the JavaScript ecosystem. We tried to counter this issue by using a selection of 1000 consumer packages for each of the different projects to open up the possibility of having a wide variety of packages.

The extent to which our study can be generalized, is known as *external validity*. Our research was carried out on open source systems. In our case, the three selected projects have their own set of rules and often focus on making their packages as small as possible. This can result in interesting constructs as shown in Listing 6.1, or removal of function parameters in favor of `argument.length`. With closed source systems there might be stricter rules or more focus on writing readable code over compact code. In such a case, the tool might perform better as its current state focuses on the function definitions available in the official JavaScript documentation.

Also, the empirical study is limited to the JavaScript programming language. This does not necessarily mean that our results can be applied to other dynamic languages, such as Python, as they can have different coding practises or dependency management.

6.4 Future Work

There are multiple suggestions for future work that we can come up with. To begin with, a similar tool could be created but focused on JavaScript projects that have type annotations (e.g. by making use of TypeScript or Flow). The results of such a study could provide interesting insights in the usability of enabling types in JavaScript.

Another option would be to improve our current approach and tool to work with frameworks such as React-Native. React-Native introduces new constructs by making use of their own JSX components, which are currently not known to our tool. Our choice for a parser to generate ASTs, was also decided with this in mind, as Flow is created by Facebook and is compatible with JSX. This makes it possible to extend our tool with such a framework.

Furthermore, an interesting aspect that we left untouched in our current research, is the importance of having good documentation regarding breaking changes. It can be interesting to conduct a qualitative study that looks into the different ways projects currently try to inform their users of breaking changes between versions. This can range from extensive documentation, special breaking changes documents or even migration guides to tools. As a result, this might further indicate the need of a generally applicable tool that is capable of detecting breaking changes.

Finally, we cover the effect of breaking changes on consumers, by looking at the version they use of a project and the corresponding breaking changes in case of an update. However, in addition to this, it can be interesting to conduct further research into the exact reasons for not updating to the latest version. Again a qualitative study could be conducted among developers to find the impact of breaking changes among other factors for not updating their project dependencies.

Chapter 7

Related work

This section contains a selection of current tools that are able to detect breaking changes in programming languages. To our knowledge, no such tools exist for JavaScript yet. Furthermore, in the subsequent sections we present multiple studies on change detection, dependencies and breaking changes that are relevant to our work.

7.1 Existing Tools

There are a number of existing tools that we could find that focus on detecting breaking changes, however most focus on the .NET language. To start, Endjin[2] looked into a way of finding breaking changes in their .NET code bases. Their aim is to automatically detect changes between updates in their code and determine the correct semantic version from that. This will help them automate the release of their packages on NuGet with the proper version. Another existing tool for .NET is LibCheck[1], which is created by Microsoft. Its focus lies on detecting changes in public interfaces between versions.

The only program that we could find that works with JavaScript and is concerned with breaking changes, is called Cracks[6]. The tool focuses on detecting breaking changes in a project, by using the available tests and comparing the results from the latest release against the current code base. When no breaking changes are introduced in the API of the project, these tests should pass. The downside of this approach is that it cannot be used to determine breaking changes in a consumer package. Furthermore, it also requires the existence of an extensive test suite that covers all of the API endpoints in a project.

Other programs we found include NDepend[8], a premium .NET tool-suite that can also detect breaking changes, ApiChange[3], RevAPI for Java[9], and Swagger for a Ruby API[10]

7.2 Change Detection

Sunghun *et al.*[16] looked into automatically identifying bug-introducing changes. This was done on two Java system; Columba and the Eclipse IDE. Their approach is based on extending previous work that made use of bug-identifiers by looking for keywords in change

log texts, or bug tracking systems if one was used. Furthermore, they run a diff tool to find differences between revision of a file, which could have introduced a bug. Their work could be used to expand our analysis, as it could provide a way to look for possible breaking changes inside function bodies, in contrast to just the function signature.

Dantas *et al.*[13] looked into the use of UML for detecting change patterns. A downside of their approach is that it requires the use of UML diagrams that need to be updated when parts of the code change. For JavaScript this is not always relevant, but since it can also be used with OOP in mind, there might be cases in which UML diagrams are being used. This would provide other ways of detecting breaking changes that are tool is currently lacking, such as looking into changes in inheritance.

7.3 Dependencies in Software Systems

Bogart *et al.*[11] interviewed developers, including two working with Node.js, on how they respond to changes to, and stability of, their dependencies. From their analysis it follows that the Node.js developers leave their trust in semantic versioning, as that is their way of knowing when breaking changes could occur (e.g. major version changes). However, in our own research we saw that this is not always adhered to and that shows the possible need of a tool that can check for breaking changes.

Cox *et al.*[12] focuses on what they called the "Dependency Freshness" in software systems. In other words, they proposed a metric that quantifies how up to date dependencies of a software system are. Their research was carried out on 75 Java system, all making use of Maven for their dependencies. They conclude that systems with a low dependency freshness score have a four times larger chance of containing security vulnerabilities. Furthermore, they stated that it was not a common practice to regularly update dependencies, as only 16.7% of the systems were up to date and over 50% of the systems were at least 5 versions behind.

Raemaekers *et al.*[18] evaluated the stability of third-party libraries, by means of four metrics. This was done by looking to Java systems and their corresponding Maven dependencies. They show for a real world example that due to expected compatibility issues, updates were deferred as long as possible. Eventually much larger effort was needed to perform the upgrade, than might be needed if done incrementally over time. This indicates that backward compatibility of a third-party library is preferred, but poses a challenge to a developer to make their API interfaces flexible for adaptations.

Somewhat similar to our work is the work by Hejderup[14], which focused on the vulnerabilities in dependencies of JavaScript projects. The outcome of the study indicates that a third of the modules they inspected, depend on a vulnerable version. They indicate breaking changes as one of the potential reasons for not updating the vulnerable dependencies.

Schoenmakers *et al.*[22] proposes an approach that evaluates the ease of upgrading a system, including an algorithm to minimize the amount of changes needed. This is done by looking at the public interfaces of the modules and the impact they will have on dependencies in other modules. Although the actual problem is NP-complete, their approximation based on heuristics is in 58% of the time able to find an optimal solution.

The effect of API deprecations on developers has been investigated by both Sawant *et al.*[21] (for 25 thousand Java Systems) and Robbes *et al.*[20] (for 2600 clients based on Smalltalk, later repeated for Pharo [15]). Although the systems are different, e.g. Java being static typed and Smalltalk being dynamic, they came to somewhat similar conclusions. Both indicate that the use of deprecations can be improved upon. Sawant *et al.* also concludes that clients that do not properly update their API versions, accumulate technical debt. This can in turn be an incentive to not update at all.[21]

7.4 Breaking Changes in Software Systems

Raemaekers *et al.* [19] explored the use of semantic versioning by looking into 7 years of Java projects on the Maven Repository. They conclude that in both minor and major API versions breaking changes are introduced, in the same amounts. This indicates that consumers can not depend on the use of semantic versioning, as breaking changes are equally likely to be introduced in versions that should not have them. Furthermore, they also concluded that the use of deprecation tags is often lacking before a function is removed.

Additionally, Raemaekers *et al.* [17] also created an entire dataset based on 148,253 Java libraries, taken from the Maven Repository. The dataset focuses on different kind of metrics and also lists breaking changes between library versions. Its main goal is to be used for further large scale research on releases and evolution of dependencies.

Xavier *et al.* [24] note that libraries are often backward incompatible, even though this can result in problems for their users. However, the exact reasons for introducing the breaking changes is not clear. Therefore they interviewed developers of popular Java libraries to find their reasons for introducing breaking changes. They show that developers are aware of the risks of them introducing breaking changes and their most frequent explanation is related to simplification. Other reasons include refactoring and bug fixes.

Chapter 8

Conclusion

We looked into the possibility of detecting breaking changes in a dynamic programming language, namely JavaScript. Furthermore we described a way to combine results of individual packages into a list of public breaking changes for a project. From the results it follows that the recall of our approach varies between 0.43 and 0.80 per project, meaning that it detects between 43% and 80% of the available breaking changes. For individual consumer packages, this it between 0.28 and 0.77.

The precision ranges between 0.22 and 0.56 for projects, indicating that the tool incorrectly lists a change as ”breaking” in 44% to 78% of the cases. We can account this to changes in the code, which based on our static analysis, falls within the rules we set for considering a change as breaking. Of these, almost 88% is due to three coding practises, including removal of private parameters, function returns and the use of `arguments.length`. For individual consumer packages, the mean precision is between 0.59 and 0.82 instead.

The results of the empirical study gives us insights in how often breaking changes occur and how many consumer packages suffer from them. Most breaking changes occur in major version changes, where one would expect them if semantic versioning is followed. However, Underscore was an exception to this, as it does not make use of major versions. The overall amount of versions that contain at least one breaking change, ranges from 1.6% to 36.3%.

Lastly, we saw that between 7.6% and 36.2% of the consumer packages are making use of the latest version of a project, indicating that the majority is at least one version behind. In addition, between 9.8% and 25.8% of consumer packages seem to suffer from breaking changes. All in all we believe that a tool for detecting breaking changes is a welcome addition for JavaScript, seeing that it can be of use to a lot of consumers. We encourage others to help with further development of the tool to increase its reliability.

Bibliography

[1] 4.4 Finding Changes Between Assembly Versions with LibCheck. https://www.safaribooksonline.com/library/view/windows-developer-power/0596527543/ch04s05.html, accessed on: 17-05-2017.

[2] An experiment to automatically detect API breaking changes in .NET assemblies and suggest a Semantic Version number. https://blogs.endjin.com/2016/02/an-experiment-to-automatically-detect-api-breaking-changes-in-dot-net-assemblies- accessed on: 17-05-2017.

[3] ApiChange, http://apichange.codeplex.com, accessed on: 17-05-2017.

[4] ASTExplorer, https://astexplorer.net, accessed on: 02-03-2017.

[5] Clirr, http://clirr.sourceforge.net, accessed on: 17-05-2017.

[6] Cracks, https://github.com/semantic-release/cracks, accessed on: 17-05-2017.

[7] How one developer just broke Node, Babel and thousands of projects in 11 lines of JavaScript, https://www.theregister.co.uk/2016/03/23/npm_left_pad_chaos/, accessed on: 28-09-2017.

[8] NDepend.API and Power Tools. https://www.ndepend.com/features/api-and-tools, accessed on: 17-05-2017.

[9] Revapi, https://revapi.org, accessed on: 17-05-2017.

[10] Using Swagger to detect breaking API changes. https://swagger.io/using-swagger-to-detect-breaking-api-changes/, accessed on: 17-05-2017.

[11] Christopher Bogart, Christian Kästner, and James Herbsleb. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *Automated Software Engineering Workshop (ASEW), 2015 30th IEEE/ACM International Conference on*, pages 86–89. IEEE, 2015.

[12] Joël Cox, Eric Bouwers, Marko van Eekelen, and Joost Visser. Measuring dependency freshness in software systems. In *Proceedings of the 37th International Conference on Software Engineering-Volume 2*, pages 109–118. IEEE Press, 2015.

[13] Cristine R Dantas, Leonardo Gresta Paulino Murta, and Cláudia Maria Lima Werner. Consistent evolution of UML models by automatic detection of change traces. In *Principles of Software Evolution, Eighth International Workshop on*, pages 144–147. IEEE, 2005.

[14] JI Hejderup. In dependencies we trust: How vulnerable are dependencies in software modules? In *MSc Thesis, TU Delft*, 2015.

[15] André Hora, Romain Robbes, Nicolas Anquetil, Anne Etien, Stéphane Ducasse, and Marco Tulio Valente. How do developers react to API evolution? the Pharo ecosystem case. In *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pages 251–260. IEEE, 2015.

[16] Sunghun Kim, Thomas Zimmermann, Kai Pan, E James Jr, et al. Automatic identification of bug-introducing changes. In *Automated Software Engineering, 2006. ASE'06. 21st IEEE/ACM International Conference on*, pages 81–90. IEEE, 2006.

[17] Steven Raemaekers, Arie van Deursen, and Joost Visser. The maven repository dataset of metrics, changes, and dependencies. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, pages 221–224. IEEE Press, 2013.

[18] Steven Raemaekers, Arie van Deursen, and Joost Visser. Measuring software library stability through historical version analysis. In *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pages 378–387. IEEE, 2012.

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

[20] Romain Robbes, Mircea Lungu, and David Röthlisberger. How do developers react to API deprecation?: the case of a Smalltalk ecosystem. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 56. ACM, 2012.

[21] Anand Ashok Sawant, Romain Robbes, and Alberto Bacchelli. On the reaction to deprecation of 25,357 clients of 4+ 1 popular Java APIs. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 400–410. IEEE, 2016.

[22] Bram Schoenmakers, Niels Van Den Broek, Istvan Nagy, Bogdan Vasilescu, and Alexander Serebrenik. Assessing the complexity of upgrading software modules. In *WCRE*, pages 433–440, 2013.

[23] Arie Van Deursen, Paul Klint, and Frank Tip. Origin tracking. *Journal of Symbolic Computation*, 15(5-6):523–545, 1993.

[24] Laerte Xavier, Andre Hora, and Marco Tulio Valente. Why do we break APIs? First answers from developers. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*, pages 392–396. IEEE, 2017.